
# ESCUELA SUPERIOR DE INGENIERÍA

## INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

DESARROLLO DE UN SISTEMA DE MONITORIZACIÓN DE BUS CAN 2.0 BASADO EN DISPOSITIVOS PROGRAMABLES DEL TIPO FPGA

Mario Márquez Luciano

25 de julio de 2011


ESCUELA SUPERIOR DE INGENIERÍA

INGENIERO TÉCNICO EN INFORMÁTICA DE SISTEMAS

# DESARROLLO DE UN SISTEMA DE MONITORIZACIÓN DE BUS CAN 2.0 BASADO EN DISPOSITIVOS PROGRAMABLES DEL TIPO FPGA


- Departamento: Ingeniería de Sistemas y Automática, Tecnología Electrónica y Electrónica
- Directores del proyecto: Mª Ángeles Cifredo Chacón, Juan Manuel Barrientos Villar
- Autor del proyecto: Mario Márquez Luciano


Cádiz, 25 de julio de 2011

Fdo: Mario Márquez Luciano

# Agradecimientos

Me gustaría dedicar este texto a mi familia por su continuo apoyo durante la realización de mis estudios y de este proyecto final de carrera, a todos los compañeros con los que he compartido estos tres años de carrera y a todos los profesores que me han dado la formación necesaria durante toda la carrera.

Especial agradecimiento a Mª de los Ángeles Cifredo Chacón, codirectora de este proyecto, por su ayuda durante la realización de este proyecto final de carrera y por prestar su coche para las pruebas del sistema de monitorización.

# Licencia

Este documento ha sido liberado bajo Licencia GFDL 1.3 (GNU Free Documentation License). Se incluyen los términos de la licencia en inglés al final del mismo.

Copyright (c) 2011 Mario Márquez Luciano.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

# Notación y formato

Cuando éste proyecto se refiera a un programa en concreto, se utilizará la notación: *Xilinx EDK*.

Cuando la referencia sea a un comando, o función de un lenguaje, se usará la notación: `rm fichero`.

# Índice general

# Indice de figuras

# Indice de tablas

# Capítulo 1

# Introducción

## 1.1. Descripción

Este documento de memoria de proyecto final de carrera describe todo el proceso que se ha seguido para el desarrollo de cada una de las cuatro fases de las que se compone:

1. Diseño de un periférico receptor de tramas basado el protocolo CAN usando un lenguaje de descripción hardware para su posterior implementación con dispositivos programables FPGA.

2. Diseño e implementación de un sistema embebido en una FPGA (Placa de evaluación), que contará con un procesador software también descrito mediante un lenguaje de descripción de hardware, el periférico CAN anteriormente descrito y otros periféricos auxiliares como un controlador para la pantalla LCD de la placa de evaluación y una UART para enviar información al exterior de la FPGA para poder comunicar el sistema embebido con una aplicación software corriendo en un PC y que será denominada *Monitoring Tool*

3. Programación del procesador del sistema embebido de la fase anterior usando lenguaje de programación C++ para hacer uso del periférico receptor de tramas y comunicarlo con el exterior.

4. Desarrollo de una aplicación software que sea capaz de comunicarse con el sistema embebido, recibir los datos que el sistema embebido de la fase anterior envíe pertenecientes a las recepciones del periférico receptor de tramas y que muestre por pantalla esos datos de una manera útil y fácil para el usuario.

## 1.2. Objetivos

Una vez conocidas las fases del proyecto, es importante establecer una serie de objetivos a alcanzar en cada una de ellas para que todo lo desarrollado en este proyecto final de carrera tenga éxito:

1. El periférico que se desarrollará presentará una interfaz estándar para que pueda ser usado en todo tipo de sistemas embebidos implementables en dispositivos FPGA, es decir, que sea reutilizable. Haciendo el periférico genérico conseguimos que cualquier diseñador de circuitos para dispositivos FPGA pueda usarlo si así lo deseara sin encontrarse con problemas de compatibilidad con el procesador que use.

2. Hacer el código del periférico de modo que sea comprensible por ingenieros en informática.

3. Demostrar que el periférico CAN es totalmente genérico, se incluirá en dos sistemas embebidos diferentes, uno con un procesador libre y otro con un procesador privativo, ambos de Xilinx Inc. que presentan diferentes interfaces de conexión con sus periféricos.

4. El programa software que se comunicará con el sistema embebido tendrá una interfaz de usuario amigable para que sea sencillo de arrancar el proceso de captura de la información así como que su visualización sea agradable para el usuario y que muestre los datos de forma que sean útiles para extraer información y conclusiones.

5. Publicar todo el software y hardware que se desarrolle en este proyecto con una licencia de software libre.

6. Hacer que el código sea accesible al mayor número de personas. Para conseguirlo, el objetivo anterior debe cumplirse. Además todo el código escrito en este proyecto final de carrera tiene como lenguaje de desarrollo el inglés para que pueda ser visualizado y modificado por la comunidad de usuarios.

## 1.3. Motivación

La principal motivación para la realización de este proyecto es poner en práctica la interrelación entre electrónica y la informática ya que durante la carrera hay pocas asignaturas que traten esta materia.
Este proyecto servirá para ampliar mis conocimientos sobre diseño de circuitos electrónicos para ser implementados en FPGAs, programación de microcontroladores, programación a bajo nivel de sistemas hardware, protocolos de comunicaciones y diseño de aplicaciones que interactúen con sistemas hardware.

Otra de las motivaciones para la realización de este proyecto es la importancia de las nuevas técnicas del codiseño HW/SW y los sistemas embebidos.

Las nuevas técnicas de diseño a nivel de sistema (ESL, Electronic System Level), están permitiendo que las ventajas del diseño con FPGAs dejen de ser patrimonio exclusivo de los ingenieros de hardware y pasen a ser una alternativa para los ingenieros de software que consiguen mejoras en el rendimiento de sus algoritmos gracias a los beneficios del paralelismo del hardware. Especialistas ajenos a los lenguajes HDL consiguen implementar sus diseños en dispositivos programables mediante los nuevos lenguajes de alto nivel (HLL) para síntesis, todo consumiendo un tiempo muy pequeño comparado con las técnicas más antiguas e incluso posibilitando la existencia de una solución hardware-software monochip (System on a chip, SoC). Disponer de

una clasificación de FPGAs simplificaría aún más la difícil tarea de colocar un producto en el mercado justo a tiempo mediante la técnica ESL.

## 1.4. Antecedentes

En el mercado de las comunicaciones CAN hay gran cantidad de dispositivos y herramientas para recibir/enviar información orientadas principalmente al mundo de la automoción. Estos dispositivos y herramientas son privativos y algunos bastante caros, además de sólo ser compatibles con ciertos modelos de coche. Es importante destacar que están diseñadas para vehículos olvidando un importante sector de mercado que es la industria donde también se usa mucho el protocolo CAN.

Centrándonos en los dispositivos FPGA, hay una serie de periféricos receptores basados en el protocolo CAN escritos en lenguaje VHDL en forjas de hardware libre pero que tienen un gran problema si es un ingeniero en informática el que quiere leer el código fuente. Ese problema es que han sido diseñados por ingenieros electrónicos siguiendo un método de diseño RTL, lo cual quiere decir que se diseña y codifica como si fueran componentes electrónicos usando funciones lógicas **"AND", "OR", etc...** lo cual es muy complicado de entender puesto que la implementación no sigue un flujo algorítmico.

## 1.5. Alcance

Con este proyecto se pretende realizar una implementación del periférico receptor de tramas CAN y el resto de componentes y software que lo acompaña de forma que sea comprensible tanto por los ingenieros en informática como por los ingenieros en electrónica. Así mismo la liberación del código fuente comentada anteriormente permitirá que usuarios que no estaban dispuestos a comprar un dispositivo y la aplicación, poder usar todo lo desarrollado en este proyecto con total libertad y poder modificarlo y adaptarlo a sus necesidades. Es muy importante destacar que los componentes implementados en este proyecto podrán ser utilizados también en entornos industriales si se deseara.

Es importante resaltar que **CAN** es un protocolo propietario perteneciente a la compañía **Robert Bosch GmbH** y que tiene una serie de patentes que hay que respetar. El desarrollo de un periférico o bloque en VHDL basado en el protocolo CAN no entra en conflicto con ninguna de las patentes del protocolo. Es más, en las forjas de harware libre como puede ser **Open Cores** hay una serie de proyectos basados en el protocolo CAN. Lo que si debe ser tenido en cuenta es que si se deseara implementar el diseño del periférico receptor de tramas escrito en VHDL en una FPGA con fines comerciales, sería necesario pagar una licencia de uso a **Robert Bosch GmbH**.

## 1.6. Estructura del documento

El primer capítulo de este documento, en el que nos encontramos actualmente, contiene los objetivos, el alcance, los antecedentes y un glosario de términos del proyecto final de carrera.

En el segundo capítulo se comentará la planificación que se ha seguido para el desarrollo de este proyecto incluyendo las fases en las que ha sido dividido y un diagrama de Gantt.

El tercer capítulo contiene una descripción de todas las tecnologías que han sido empleadas durante este proyecto desde las FPGAs hasta el protocolo de comunicaciones CAN.

El cuarto capítulo es una descripción general del proyecto.

El quinto capítulo trata sobre la especificación de los requisitos del sistema.

Los cinco siguientes capítulos describen todo el proceso de desarrollo de cada una de las fases de este proyecto final de carrera.

El sexto capítulo trata sobre el periférico receptor escrito en VHDL, el séptimo capítulo sobre el diseño de un sistema embebido con núcleo PicoBlaze, el octavo sobre el diseño de un sistema embebido con núcleo MicroBlaze, el noveno sobre todo el proceso de ingeniería del software que se ha seguido para desarrollar un sistema de control para el sistema embebido del capítulo sexto. Por último el décimo capítulo trata sobre el desarrollo de una aplicación de monitorización para ordenadores personales que sea capaz de comunicarse con el sistema embebido de los capítulos anteriores siguiendo el proceso de ingeniería del software orientada a objetos.

En el undécimo capítulo se describe la puesta en marcha del sistema que ha sido desarrollado durante este proyecto final de carrera.

En el último capítulo se incluyen las conclusiones y valoraciones tanto personales como técnicas. Además se incluyen las posibilidades de ampliación.

Además se incluyen distintos apéndices tales como el manual de instalación, el manual de usuario y la bibliografía.

Al ser un producto de Software Libre se incluye la licencia en la que se basa la documentación del proyecto.

## 1.7. Definiciones, acrónimos y abreviaturas

- FPGA: Field Programmable Gate Array. Dispositivo semiconductor que contiene bloques lógicos configurables mediante un lenguaje de programación especializado.
- VHDL: Representa la combinación de los acrónimos VHSIC y HDL.
- VHSIC: Very High Speed Integrated Circuit (Circuito integrado de muy alta velocidad).
- HDL: Hardware Description Language (Lenguaje de descripción de hardware).
- IEEE: Institute of Electrical and Electronic Engineers (Instituto de ingenieros eléctricos y electrónicos).

- Soft Core: Microprocesador software que implementado sobre un dispositivo programable FPGA se comporta como lo haría uno hardware.
- Embeded System: Un sistema embebido es un sistema diseñado para realizar una o varias funcionalidades. Se caracteriza por tener la memoria, el procesador y los periféricos encapsulados en el mismo chip o placa.
- Picoblaze: Soft Core de 8 bits libre desarrollado por la empresa Xilinx.
- Microblaze: Soft Core de 32 bits propietario desarrollado por la empresa Xilinx.
- CRC: Comprobación de redundancia cíclica. Es una función que recibe un flujo de datos de cualquier longitud como entrada y devuelve un valor de longitud fija como salida.
- CAN: Controller Area Network. Protocolo de comunicaciones muy usado en automoción e industria. Es la base de este proyecto.
- Bit Stuffing: Técnica usada en telecomunicaciones donde se insertan bits que no contienen información a la transmisión para el control de errores y sincronización.
- UART: Universal Asynchronous Receiver-Transmitter. Es el hardware que se encarga de las comunicaciones seriales ya sean las que estén basadas en el protocolo RS-232 como otros protocolos de comunicación serie.
- ECU: Engine Control Unit. Centralita encargada de controlar los parámetros de un motor de combustión.
- DCM: Digital Clock Manager. Componente presente en las FPGA de Xilinx que permiten cambiar la frecuencia y fase de una señal de reloj base.
- CSMA/CD+AMP: Carrier Sense Multiple Access with Collision Detection and Arbitration Message Priority

# Capítulo 2

# Planificación

## 2.1. Fases del proyecto

Como comentamos en la introducción, este proyecto tiene cuatro fases o partes bien diferenciadas. A continuación vamos a comentar el proceso de desarrollo que se ha seguido en cada una de ellas.

### 2.1.1. Fase de desarrollo del periférico receptor CAN

#### Análisis del periférico

Se realizó un análisis profundo del protocolo de comunicaciones CAN siguiendo las especificaciones del fabricante para ver realmente como funciona este protocolo de comunicaciones y qué elementos necesitamos a la hora de crear desde cero nuestro periférico receptor de tramas CAN. Se analizan también las frecuencias y velocidades de trabajo del protocolo para adaptarlas al periférico.

#### Diseño del periférico

Una vez tenemos el análisis completo de los componentes del periférico, en esta fase se define la interconexión de todos los componentes que lo componen y como interactúan entre sí para conseguir el objetivo de recibir tramas CAN. También se incluye la interfaz con el exterior.

#### Implementación

Es la fase más larga y más compleja de las que se han realizado con el periférico. Consistió en codificar el diseño del periférico con un lenguaje de descripción de hardware.

#### Pruebas

Se realizaron en paralelo con la implementación. A medida que se iban desarrollando los componentes del periférico se simulaban con el software de simulación dándole estímulos a las señales de entrada para visualizar las salidas y comprobar su corrección. Además del simulador se probaron directamente sobre la placa de evaluación mediante la descarga y la configuración

de la FPGA que contiene viendo que el comportamiento del simulador se mantenía en el mundo real. Finalmente se prueba la integración de todos los componentes con el simulador y con la placa de evaluación. Se usaron dos placas de evaluación, una con el periférico implementado y otra con un generador de tramas también diseñado en este proyecto para depurar y probar los diseños, ambas interconectadas entre sí. Además de la prueba con las dos placas de evaluación Spartan 3E starter kit, se utilizado un microcontrolador LPC32-78 de NPX Semiconductors que entre uno de sus mucho periféricos, posee un módulo CAN con el que podemos enviar tramas al periférico que de ha desarrollado para depurarlo.

### 2.1.2. Fase de desarrollo del sistema embebido con procesador Picoblaze

#### Análisis del sistema embebido

Se realizó un análisis exhaustivo de las necesidades que requería nuestro sistema embebido con núcleo **"Picoblaze"** para que hiciera un uso adecuado del periférico y mostrara sus características. Así mismo se analizan todos los componentes del sistema embebido: procesador, memoria y periféricos.

#### Diseño del sistema embebido

Una vez ha sido realizado el análisis, en el diseño se define la interconexión de los componentes del sistema embebido, la interfaz de conexión del periférico CAN al bus del procesador usando multiplexores y la comunicación con el exterior usando la UART.

#### Implementación

La implementación para cada procesador tuvo dos partes:

1. Codificación de la interfaz del periférico CAN con el bus del procesador usando multiplexores.

2. Implementación del sistema embebido. Para la implementación con el procesador **"Picoblaze"**, para el cual Xilinx proporciona su código VHDL, se usará el entorno de desarrollo para dispositivos FPGA de Xilinx (Xilinx Project Navigator) así como un entorno para programar en ensamblador su ROM.

#### Pruebas

Las pruebas de funcionamiento del sistema embebido fue realizada directamente sobre la placa de evaluación. Se utilizó un periférico UART para enviar los datos que procesa el sistema embebido para poder ver los resultados en un terminal de un PC.

### 2.1.3. Fase de desarrollo del sistema embebido con procesador Microblaze

#### Análisis del sistema embebido

Se realizó un análisis exhaustivo de las necesidades que requería nuestro sistema embebido para que hiciera un uso adecuado del periférico y mostrara sus características. Así mismo se analizan todos los componentes del sistema embebido: procesador, memoria y periféricos. Esta fase se realizó para el procesador **"Microblaze"**. Se analizan también los requisitos de velocidad, frecuencia y espacio del sistema embebido.

#### Diseño del sistema embebido

Una vez ha sido realizado el análisis, en el diseño se define la interconexión de los componentes del sistema embebido, la interfaz de conexión del periférico CAN al bus del procesador y el mapa de memoria del sistema embebido.

#### Implementación

La implementación para cada procesador tuvo dos partes:

1. Codificación de la interfaz del periférico CAN con el bus del procesador.
2. Implementación del sistema embebido. Para implementarlo con el procesador **"Microblaze"**, el cual es privativo, se usó su propio entorno de desarrollo (Xilinx EDK).

#### Pruebas

Las pruebas de funcionamiento del sistema embebido fue realizada directamente sobre la placa de evaluación. Se utilizó un periférico UART para enviar los datos que procesa el sistema embebido para poder ver los resultados en un terminal.

### 2.1.4. Fase de desarrollo del software para el microprocesador del sistema embebido

#### Análisis software

Teniendo ya el sistema embebido implementado, se analizaron las necesidades de funcionamiento del sistema embebido con respecto a qué hacer con la información que se recibe del periférico CAN. Se analizó también el protocolo de comunicación serie que se iba a establecer con la aplicación de visualización de datos.

#### Diseño software

Siguiendo los requisitos de análisis, se procedió a desarrollar todo el modelo de diseño de software.

#### Implementación

En base a las especificaciones de diseño se procedió a codificar el software del procesador usando lenguaje C.

#### Pruebas

Se realizaron las pruebas de funcionamiento sobre la placa de evaluación.

### 2.1.5. Fase de desarrollo de la aplicación software

#### Análisis software

Se procedió a analizar los requisitos para establecer la comunicación por interfaz serie entre una aplicación software y un sistema embebido. Se prestó atención a encontrar y describir los objetos existentes en el dominio del problema y la definición del protocolo de comunicación con el sistema embebido, el cual es el mismo que el del software del procesador.

#### Diseño software

Siguiendo los requisitos de análisis, se procedió a desarrollar todo el modelo de diseño de software. Así mismo se procedió al diseño de la interfaz de usuario.

#### Implementación

En base a las especificaciones de diseño de codificó la aplicación haciendo uso del lenguaje C++.

#### Pruebas

Las pruebas se realizaron tanto en un sistema Linux como en uno Windows estableciendo una comunicación por la interfaz serie entre la aplicación y el sistema embebido completamente implementado con el periférico CAN y el procesador corriendo el software comunicación.

## 2.2. Diagrama de Gantt

A continuación se muestra el diagrama de Gantt del proyecto que incluye todas las fases en las que ha consistido. Como se puede observar, la duración total ha sido de 10 meses comenzando en octubre de 2010 y acabando a mediados de julio de 2011.

Figura 2.1: Diagrama de Gannt 1

Figura 2.2: Diagrama de Gannt 2

# Capítulo 3

# Descripción de tecnologías

En este capítulo se va describir todos los conceptos teóricos y técnicos necesarios para la comprensión del funcionamiento del dispositivo que se va a desarrollar en este proyecto final de carrera.

## 3.1. FPGAs

Una FPGA o Field Programmable Gate Array es un circuito integrado que contiene bloques que son capaces de implementar funciones lógicas y programables por parte del usuario mediante un lenguaje de descripción de hardaware. Cada uno de esos bloques está conectado mediante interconexiones programables por parte del usuario.

#### Antecedentes históricos

El comienzo de la industria de las FPGA tiene lugar en la década de los ochenta. Las FPGAs que conocemos hoy día, surgen de los experimentos de Ross Freeman y Bernard Vonderschmitt, ambos fundadores de Xilinx. Ambos desarrollaron en 1985 la primera FPGA de la historia, la XC2064. Las FPGAs surgen como una evolución de la tecnología CPLD y PAL.

Los años noventa fueron el periodo de consagración de la tecnología FPGAs en el mercado y de su evolución tanto en tecnología como en tamaño. Las FPGAs comenzaron a usarse en diversos campos como las telecomunicaciones o la automoción.
En la actualidad las compañias Xilinx y Altera copan el mercado, siendo la primera la más potente con un 51,2 % de las FPGAs del mundo.

#### Recursos de una FPGA

Las FPGA están compuestas por tres recursos principales y otros adicionales:

- Bloques de lógica
- Recursos de interconexión
- Bloques de entrada/salida.
- Recuros adicionales (Multiplicadores, memoria embebida o DSP).

Los bloques de lógica son aquellos que pueden ser configurados con una determinada función lógica. Normalmente, en una FPGA, los bloques están compuestos por unos elementos denominados Slices. A su vez, un Slice está compuesto por celdas lógicas, las cuales son definidas a continuación: Las celdas lógicas están compuestas de los siguientes elementos:

- LUT.
- Multiplexores.
- Biestables.

Una LUT es una tabla de valores preasignados. Durante la programación, la LUT almacena los valores de todas las posibles combinaciones de entrada para la función lógica programada, asignándole a cada uno de ellos una dirección.

En cuanto a los biestables, se usan para almacenar las salidas de cada slice para que el dato que se transmita entre ellos sea estable.

### Características

La principal característica de una FPGA es la capacidad de ser reprogramable, de modo que permite al programador revisar y mejorar los circuitos electrónicos que se van a implementar. Esto contrasta con los ASIC que sólo pueden ser programados una vez. Los ASIC son confeccionados a partir de una máscara fija que los dota de una funcionalidad exclusiva. Cualquier mejora o modificación, por simple que sea, obliga a fabricar de nuevo el ASIC, lo que supone un elevado coste y demora de su salida al mercado.

La programación de una FPGA se realiza a nivel de interconexiones de bloques de lógica.

### Programación de las FPGA

La programación de una FPGA puede ser realizada mediante leguajes de descripción de hardware (HDL) o con esquemáticos [1]. Lo normal es usar los lenguajes de descripción de hardware debido a que evitan tener que dibujar todos y cada uno de los componentes del diseño. Además, las herramientas de diseño son capaces de generar visualizaciones de los circuitos a partir de descripciones en HDL. Los lenguajes de descripción de hardware más usados en la actualidad son Verilog y VHDL. A VHDL se le dedicará una sección más adelante.
Los pasos que se siguen en el desarrollo con dispositivos programables FPGA son los que siguen:

- Realizar la descripción del circuito digital mediante un lenguaje de descripción de hardware (HDL).
- Realizar simulaciones funcionales del circuito digital.
- Generar la "netlist" en la que se describen todas las conexiones de la descripción del circuito con los bloques E/S de la FPGA.

[1] VHDL: IEEE std 1076 Verilog: IEEE std 1364

- Mediante la herramienta de automatización del fabricante, realizar los siguientes pasos:
  - Síntesis.
  - Traducción.
  - Mapeado tecnológico.
  - Emplazamiento y ruteado.
  - Generar fichero binario de configuración de tipo *.bit.
- Cargar el fichero de configuración en la FPGA mediante la herramienta que proporciona el fabricante.

Además de todo lo anterior, el programador puede simular su diseño mediante simuladores tanto del fabricante como de terceras compañias para comprobar la correcta funcionalidad de su diseño.

**Ventajas e inconvenientes de las FPGA**

Las FPGA tienen múltiples ventajas con respecto a los circuitos ASIC.

- Rápida disponibilidad: El prototipado se puede realizar en un corto período de tiempo.
- Herramientas de bajo coste: sólo es necesario el software del fabricante para implementar el circuito y descargarlo en la FPGA.
- Diseños más agresivos: la capacidad de ser reprogramables influye en el desarrollo de diseños más agresivos debido a que no se inutiliza el dispositivo una vez programado el diseño y por tanto, diseños menos eficientes.
- Verificación del diseño más efectiva: las herramientas de automatización del proceso de desarrollo favorecen comprobaciones de la corrección del diseño de manera más rápida y efectiva.
- Procesamiento paralelo.
- Desarrollos más económicos.

Como inconvenientes, presentan los siguientes:

- Mayor tamaño que otras tecnologías.
- Más lentas que otras tecnologías.
- Pueden inducir procesos de desarrollo basados en ensayo-error debido su reprogramabilidad.
- Tradicionalmente han tenido un mayor coste, aunque la situación está cambiando y cada vez son más baratas.

**Aplicaciones**

Las FPGA están siendo utilizadas en diversos campos entre los cuales podemos destacar:

- Procesamiento digital de señales
- Telecomunicaciones
- Sector aeroespacial
- Sistemas de defensa
- Automoción
- Prototipado de circuitos ASIC
- Reconocimiento del habla
- Criptografía
- Emulación de hardware

Todos los sectores anteriores necesitan gran capacidad de procesamiento paralelo además de capacidad de reconfiguración. Ambas son las principales características de las FPGA y de ahí su auge en estos últimos veinte años.


Figura 3.1: FPGA de Xilinx

## 3.2. Lenguaje VHDL

VHDL, acrónimo de VHSIC (Very High Speed Integrated Circuit) y HDL (Hardware Description Language) es un lenguaje de descripción de hardware definido por IEEE usado para la

descripción de circuitos digitales. La gran mayoría de esos circuitos digitales serán luego implementados sobre FPGA aunque el lenguaje también se usa para describir de forma genérica sin que sea necesario luego implementarlo sobre una FPGA.

La principal ventaja del VHDL frente a otros métodos de descripción de circuitos digitales es la sencilla sintaxis y facilidad de diseño jerárquico a diferencia del uso de esquemáticos los cuales son inviables cuando el tamaño del diseño es considerable.

### 3.2.1. Formas de diseño

En VHDL podemos describir los circuitos digitales de tres formas distintas:

- Comportamiento: describe el comportamiento del circuito digital. Es la forma más parecida a los lenguajes de programación software debido que se basa en sentencias secuenciales dentro de un proceso. Las sentencias se ejecutan de forma secuencial mientras que los procesos son paralelos entre sí.
- Flujo de datos: se usan sentencias de asignación que se ejecutan concurrentemente.
- Estructural: se usa para describir jerarquías entre los dos tipos anteriores. Es la que permite interconectar bloques entre sí.
- Las tres formas anteriores se pueden combinar entre sí.

### 3.2.2. Características

El lenguaje VHDL posee similares características a los lenguajes de programación estructurados. Incluye operadores lógicos, asignaciones y nuevas sentencias propias del lenguaje.

#### Biblioteca

Una biblioteca VHDL es un lugar donde se almacena toda la información relacionada con el diseño que se esté realizando. La biblioteca por defecto de un proyecto se denomina “work”. Una de las bibliotecas más importantes a usar con VHDL es la de IEEE que normalmente se incluye en los diseños.

#### Paquetes

En los paquetes se almacena información sobre tipos, objetos que pueden ser usados en los diseños.

### 3.2.3. Herramientas

Hay disponible una gran cantidad de herramientas para realizar diseños de circuitos digitales con VHDL entre las que destacan:

- Xilinx ISE
- Altera Quartus II
- Actel Libero IDE
- Lattice Diamond Design
- Labview
- Matlab
- Simulador GHDL
- Modelsim
- Isim

## 3.3. Placa de evaluación Spartan-3E Starter Kit

La Spartan-3E Starter Kit es una placa de evaluación de Digilent, uno de los mayores fabricantes de placas de evaluación basadas en FPGA y microcontroladores. La placa Spartan 3E starter kit presenta los siguientes recursos:

- FPGA Xilinx Spartan 3E FPGA de 500000 puertas lógicas. Esta FPGA tiene un encapsulado FG320 y la velocidad de trabajo es -4, siendo ésta la velocidad lenta dentro de las FPGA de Xilinx.
- Conector de 100 pins Hirose FX2 para usar como entradas y salidas.
- Conector DB15HD VGA para disponer de salida VGA para conectar a monitores compatibles VGA.
- Conector PS/2 para conectar un teclado o un ratón.
- Dos conectores DB9, uno DCE y otro DTE para comunicaciones siguiendo el protocolo RS-232.
- Conector RJ-45 Ethernet.
- Pantalla LCD
- Conector SMA para conectar un cristal de reloj externo.
- Un Potenciómetro.

- Cuatro interruptores y cuatro pulsadores.
- 8 leds.

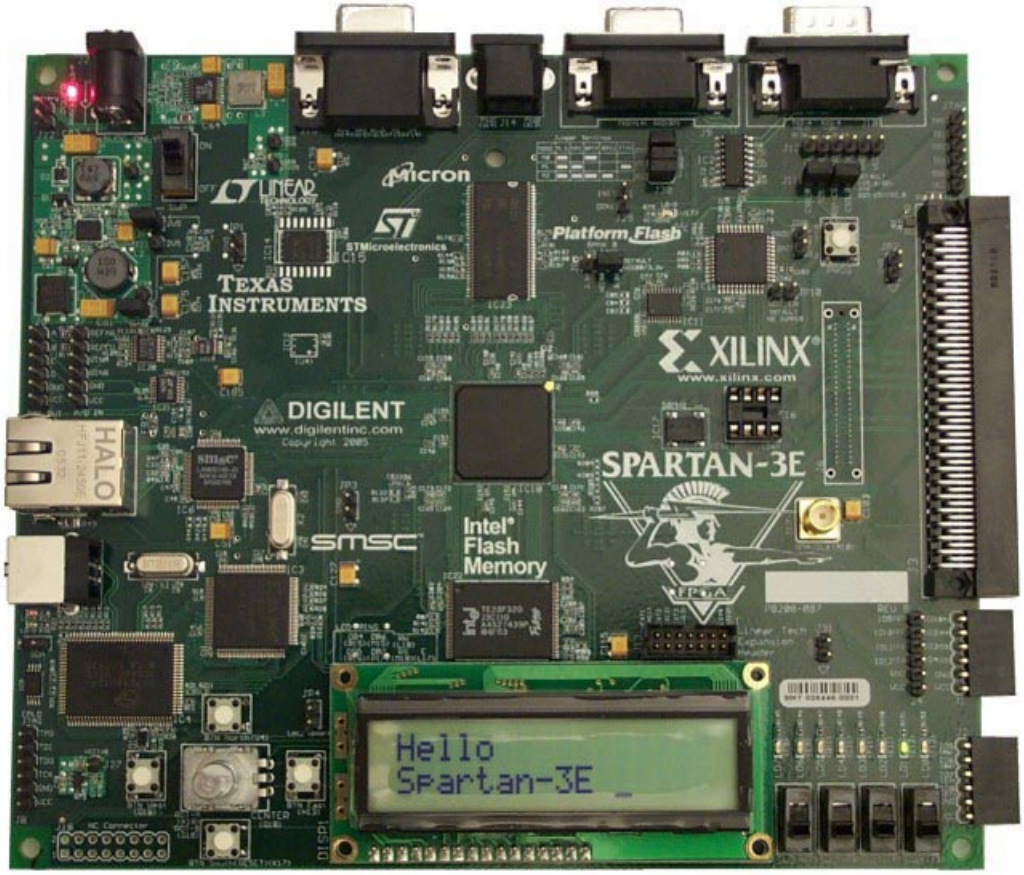


Figura 3.2: Placa de evaluación Spartan 3E (Autor: Digilent y Xilinx)

## 3.4. Placa de evaluación Olimex LPC-2378STK

La Olimex LPC-2378STK es una placa de evaluación de la empresa búlgara Olimex la cual incorpora un microcontrolador LPC2378 con núcleo ARM de NPX Semiconductors, división de microcontroladores de Philips. Presenta los siguientes recursos:

- Microcontrolador LPC2378 de NPX.
- Conector RJ-45 Ethernet.
- Conector USB.
- Dos módulos CAN
- Dos UARTS para el protocolo RS232

- Pantalla LCD de 128x128 pixels.
- Pulsadores, joystick y un potenciómetro.

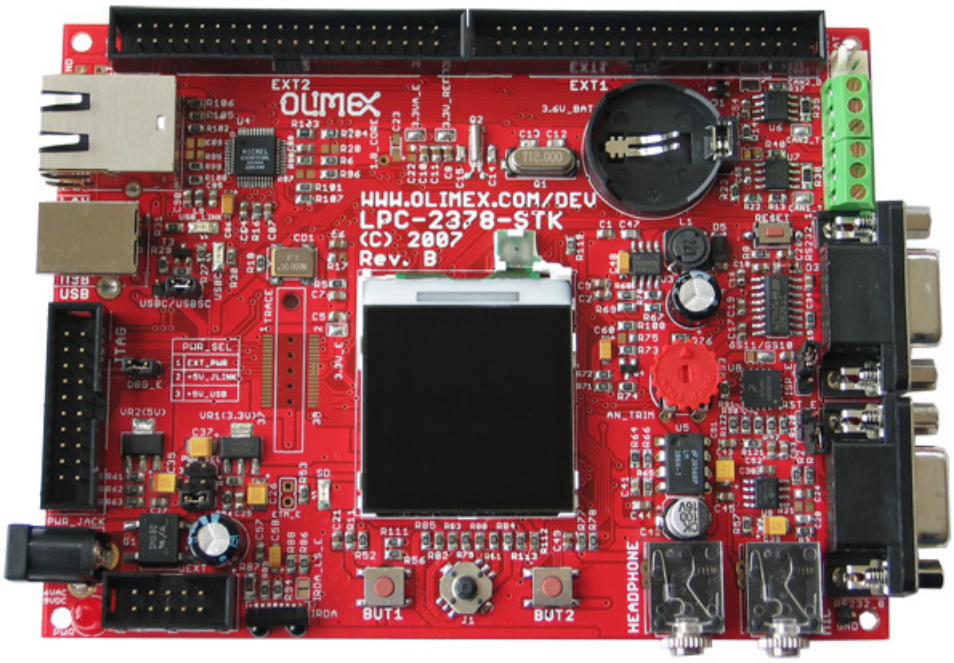


Figura 3.3: Placa de evaluación Olimex LPC-2378STK (Autor: Olimex)

## 3.5. CAN-bus

### 3.5.1. Antecedentes históricos

CAN-bus (Controller-area Network) es un protocolo de comunicaciones basado en una topología bus desarrollado para su uso en vehículos.

Sus orígenes datan del año 1983 cuando la compañía alemana Robert Bosch GmbH comenzó con el desarrollo del protocolo que finalmente fue presentado oficialmente en un congreso de la Sociedad de Ingenieros Automotrices en la ciudad de Detroit en 1986.

Los primeros controladores CAN desarrollados por los gigantes de la electrónica como Philips e Intel aparecieron un año más tarde. Robert Bosch GmbH presentó en 1991 la especificación 2.0 del CAN-bus.

Desde el año 1996 es obligatorio en Estados Unidos, siendo obligatorio en Europa desde 2001 para los vehículos de gasolina y desde 2004 para todos los vehículos diesel.

Dentro de los vehículos este bus establece una comunicación multiplexada entre las diferentes unidades de control intercambiando información sobre seguridad, confort y todo el sistema de tracción.

### 3.5.2. Tecnología

CAN está diseñado para la transmisión de mensajes en entornos distribuidos entre ECUs (Electronic Control Unit) que son las encargadas de controlar el funcionamiento de todos los dispositivos internos del automóvil. Los mensajes tienen un identificador que los hacen únicos dentro de la red de nodos. Cado nodo decide si acepta o no el mensaje dependiendo del identificador.

#### Características

- Es un sistema robusto en cuanto a la fiabilidad de transmisión de datos.
- Permite establecer prioridad a cada mensaje a enviar.
- Detección de errores.
- Retransmisión de tramas defectuosas.

#### Tipos de redes CAN

Distinguimos dos tipos de redes:

*Alta velocidad*

- Hasta 1 Mb/s de velocidad de transferencia.
- Regulada por el estándar ISO 11898-2
- Dedicada al control del motor.
- Control de las ECUs.

*Baja velocidad*

- Hasta 125 Kb/s de velocidad de transferencia.
- Regulada por el estándar ISO 11519-2/ISO 11898-3
- Dedicada al control de dispotivos electrónicos del automóvil como puertas, luces, etc...

#### Estructura y topología

El bus CAN es de tipo diferencial, lo que quiere decir que la información es transmitida por dos hilos. Todos los nodos conectados al bus reciben las tramas, lo que se denomina bus multimaestro. La comunicación es half-duplex. Se establece un sistema de arbitraje para evitar colisiones entre tramas en base al identificador de la trama.

### 3.5.3. Capas del modelo OSI

El bus CAN opera en las dos capas más bajas del modelo OSI.

### Capa física

Define los aspectos del medio físico para la transmisión de información. El bus CAN se compone de dos hilos paralelos con el correspondiente blindaje. Al final del bus se colocan unas resistencias de terminación de bus.
Los niveles lógicos son:

- Recesivo: voltaje diferencial de 0V. Es el 1 lógico.
- Dominante: voltaje diferencial entre 1,5 y 3 V. Es el 0 lógico.

La ventaja de la señal diferencial es que el ruido afecta de igual manera a ambos hilos, por lo que prácticamente no va a afectar a la señal final.

El estándar establece que la transmisión es asíncrona, es decir, cada nodo posee su propio reloj. Se utiliza codificación NRZ-I por lo que la señal permanece constante todo el tiempo que dura el bit. De todos modos, la señal es muestreada 12 veces cada bit según la especificación de Bosch para mayor fiabilidad.

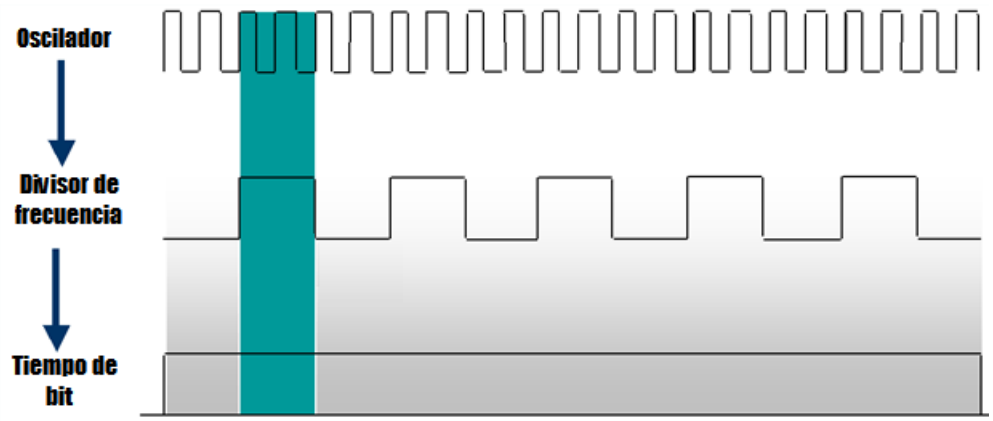


Figura 3.4: Tiempo de bit (Autor: Softing)

Debido a la codificación NRZ-I, se lleva a cabo la práctica del bit stuffing que consiste en insertar un bit de polaridad inversa después de cinco bits con la misma polaridad. El receptor de la trama posee la capacidad de detectar los bits añadidos por el transmisor y desecharlos.

### Capa de enlace

Establece el control de acceso al medio y la estructura de las tramas. Define las tareas independientes del método de acceso al medio que es muy importante en redes multimaestro puesto que todo nodo activo tiene los derechos para controlar la red y acaparar los recursos.

Cuando un nodo de la red CAN quiere enviar información a través de la red CAN puede ocurrir un intento de transmisión simultáneo entre varios nodos. El protocolo CAN lo resuelve asignando prioridades con el identificador de cada mensaje no modificable dinámicamente. El identificador más bajo es el que tiene mayor prioridad.

El método de acceso al medio es el de Acceso Múltiple por Detección de Portadora, con Detección de Colisiones y Arbitraje por Prioridad de Mensaje (CSMA/CD+AMP). Siguiendo este método, un nodo que quiera iniciar una transmisión debe esperar a que el bus esté libre. Una vez esté libre el bus comienza a transmitir el bit de inicio. Cada nodo lee del bus un bit en cada ciclo de reloj durante la transmisión de la trama y comparan el valor transmitido con el valor recibido. Mientras los valores sean idénticos, significa que el identificador que se está transmitiendo puede que sea el suyo por lo que el nodo continúa con la transmisión. En caso de detectarse una diferencia en los valores de los bits, el identificador ya no es el suyo y el nodo se retira de la transmisión hasta encontrar de nuevo el bus libre.

### 3.5.4. Tramas

Hay cuatro tipos de tramas CAN: Tramas con identificador estándar, con identificador extendido, de error y distribuidas.

#### Trama con identificador estándar

Es el único tipo de trama que envía datos desde un nodo al resto. La trama presenta la siguiente estructura:

| Nombre del campo | Logitud en bits | Descripción |
|---|---|---|
| Comienzo de trama | 1 | Indica el comienzo de la trama.<br>Debe ser dominante. |
| Identificador | 11 | Es el identificador del mensaje. |
| Trama remota | 1 | Indica si la trama es de tipo remoto. |
| Bit de identificador extendido | 1 | Indica si la trama tiene identificador extendido. |
| Bit Reservado | 1 | Se acepta tanto dominante como recesivo. |
| Longitud datos | 4 | Indica la longitud en Bytes del campo de datos. |
| Campo de datos | 0-64 | Datos enviados. |
| CRC | 15 | Comprobación de redundancia cíclica. |
| Delimitador CRC | 1 | Delimita el campo CRC. Recesivo |
| Acuse de recibo | 1 | El trasmisor manda recesivo.<br>El receptor contesta dominante. |
| Delimitador acuse de recibo | 1 | Debe ser recesivo. |
| Final de trama | 7 | Debe ser recesivo. |

Tabla 3.1: Campos de la trama estándar

#### Trama con identificador extendido

Identica a la anterior excepto que el campo identificador tiene una longitud 29 bits.

| Nombre del campo | Logitud en bits | Descripción |
|---|---|---|
| Comienzo de trama | 1 | Indica el comienzo de la trama.<br>Debe ser dominante. |
| Identificador | 11 | Es el identificador del mensaje. |
| SSR | 1 | Recesivo. |
| Bit de identificador extendido | 1 | Indica si la trama tiene identificador extendido. |
| Identificador extendido | 18 | Extensión del identificador. |
| Trama remota | 1 | Indica si la trama es de tipo remoto. |
| Bit Reservado | 2 | Se acepta tanto dominante como recesivo. |
| Longitud datos | 4 | Indica la longitud en Bytes del campo de datos. |
| Campo de datos | 0-64 | Datos enviados. |
| CRC | 15 | Comprobación de redundancia cíclica. |
| Delimitador CRC | 1 | Delimita el campo CRC. Recesivo |
| Acuse de recibo | 1 | El trasmisor manda recesivo.<br>El receptor contesta dominante. |
| Delimitador acuse de recibo | 1 | Debe ser recesivo. |
| Final de trama | 7 | Debe ser recesivo. |

Tabla 3.2: Campos de la trama con identificador extendido

#### Trama remota

Es un tipo de trama que se utiliza para pedir datos a un nodo. Como se puede observar en la tabla no posee ningún campo para enviar datos.
Tiene la estructura de una de las dos tramas anteriores pero sin el campo de longitud de datos y los datos.

#### Trama de error

Las tramas de error son enviadas de inmediato al detectar un error en la transmisión. Su estructura es la siguiente:

| Nombre del campo | Logitud en bits | Descripción |
|---|---|---|
| Banderas de error | 6-12 | Bits dominantes y recesivos de los diferentes nodos. |
| Delimitador de error | 8 | Debe ser recesivo. |

Tabla 3.3: Campos de la trama de error

### 3.5.5. Aplicaciones

El bus CAN puede ser usado en todo ámbito que necesite control distribuido en tiempo real y que además requiera robustez y fiabilidad en las comunicaciones. El ámbito donde más es usado es en la automoción.

El usar un bus de comunicaciones para la interconexión de los dispositivos electrónicos dentro de un automóvil tiene sus ventajas puesto que se elimina el cableado punto a punto con lo que se reduce peso y costes. Además la robustez frente al ruido que proporciona es muy adecuada para las comunicaciones entre las centralitas del motor que controlan la inyección electrónica, en las que un fallo en la comunicación no es tolerable.

# Capítulo 4

# Descripción general del proyecto

## 4.1. Perspectiva del producto

El proyecto trata de ofrecer un completo sistema de monitorización del bus CAN 2.0 usado en todos los vehículos a motor modernos desde hace menos de quince años.

Las cuatro fases de las que se compone el proyecto (periférico receptor VHDL, sistema embebido, software de control para el sistema embebido y software de monitorización) se integran para formar el completo sistema de monitorización.

El periférico receptor escrito en VHDL se encargará de recibir las tramas con el formato del protocolo CAN.

El sistema embebido se encargará de interactuar con el anterior receptor y comunicarse con el exterior. Esta comunicación con el exterior se hace de dos maneras bien diferenciadas:

- Pantalla LCD y diodos led de la placa de evaluación donde será implementado el sistema embebido: esta forma de visualizar los datos dotará al sistema de independencia con respecto a la aplicación de monitorización que también será desarrollada en este proyecto. El usuario podrá visualizar las tramas recibidas en la pantalla LCD y observar que el sistema está funcionando correctamente mediante los leds que indicarán en todo momento el estado del sistema. Mediante interruptores, el usuario podrá cambiar los modos de funcionamiento del sistema que son dos:

  - Modo Visualizar: En este modo la información sólo será mostrada por la pantalla LCD del sistema.
  - Modo Transmisión de datos: En este modo la información se enviará a la aplicación externa de monitorización.

- Aplicación de monitorización: El sistema embebido enviará la información que reciba a la aplicación de monitorización usando para ello el puerto serie. Para comunicar el sistema embebido con el software de monitorización mediante el puerto serie será desarrollado un protocolo de comunicaciones para controlar el flujo de información.

La aplicación de control del sistema embebido será desarrollada para el microprocesador Microblaze en C++. El código será únicamente compatible con este procesador debido a que usa funciones específicas de Xilinx para controlar los periféricos del sistema embebido. La estructura del código, sin embargo, ayuda a que su implementación en otros sistemas se pueda realizar con bastante facilidad debido a que sólo sería necesario cambiar las funciones de los componentes que sean necesarias.

Además, el software desarrollado para el control los distintos periféricos del sistema, como la UART, la pantalla LCD, etc... forma una pequeña biblioteca para periféricos de sistemas embebidos de Xilinx. Debido a esto el código puede ser en gran medida reutilizado.

Por último, la aplicación de monitorización será desarrollada con una visión hacia la facilidad de uso por parte del usuario y a mostrar los datos de una manera útil y sencilla para el usuario. La aplicación se comunicará con el sistema embebido anteriormente descrito y recibirá datos de éste para mostrarlos por pantalla e incluso almacenar un log de toda la información recibida. Esta aplicación será desarrollada tanto para sistemas Windows como GNU/Linux por lo que se ha optado por Qt para su interfaz gráfica debido a que es multiplataforma.

## 4.2. Limitaciones de memoria

### 4.2.1. Sistema embebido

La cantidad de memoria de la que se dispone en el sistema embebido es bastante escasa debido a que en gran medida depende de la FPGA donde se implementará el sistema. En la que se usará en este proyecto, disponemos sólo de 32 KiloBytes de memoria por lo que todo el desarrollo dependerá en gran medida de conseguir implementar toda la funcionalidad que se desea que tenga el sistema y además hacer que quepa en los 32 KBytes.

### 4.2.2. Aplicación de monitorización

La memoria no es un problema para la aplicación, ya que no necesita almacenar demasiada información. El único requisito de memoria a tener en cuenta es el tamaño que ocupará el fichero de log (puede llegar a ocupar centenares de MBytes) que en gran medida dependerá de la cantidad de tiempo recibiendo información del sistema embebido.

## 4.3. Limitaciones de rendimiento

El rendimiento del sistema es un parámetro crítico que depende de varios factores:

En el sistema embebido el rendimiento dependerá de la frecuencia de trabajo del microprocesador, configurable desde 50 a 75 MHz con el hardware que se usa en este proyecto y de la velocidad a la que ha sido configurada la UART. A mayor velocidad de la UART, mayor cantidad de información se enviará a la aplicación de monitorización y por tanto el número de tramas recibidas será mayor.

En la aplicación de monitorización la UART debe ser configurada a la misma velocidad que la del sistema embebido. Por tanto el rendimiento dependerá de la frecuencia de la CPU y de las funciones usadas en la aplicación. Si se usa la función de log sin ningún visualizar activo, el rendimiento será el máximo. A mayor número de visualizadores activos, el rendimiento decrece.

## 4.4. Dependencias

El sistema desarrollado en este proyecto presenta las siguientes dependencias:

### 4.4.1. Sistema embebido

El sistema embebido, debido a que usará componentes del fabricante Xilinx, dependerá de bibliotecas y componentes de Xilinx.

### 4.4.2. Aplicación de monitorización

- C++
- Biblioteca Qt
- Biblioteca Qextserialport
- PC con puerto serie. En caso de no disponer de uno es posible usar un conversor USB-RS232

# Capítulo 5

# Especificación de los requisitos del sistema

## 5.1. Periférico receptor de tramas CAN

### 5.1.1. Requisitos de interfaces externas

- El periférico debe tener una interfaz genérica que pueda ser adaptada a todo tipo de procesadores y sistemas embebidos.

### 5.1.2. Requisitos funcionales

- El periférico debe ser capaz de recibir tramas CAN de tipo estándar.
- El periférico debe ser capaz de recibir tramas CAN de tipo identificador extendido.
- El periférico debe ser capaz de rechazar tramas erróneas.
- El periférico estará adaptado al estándar CAN que define una velocidad de transmisión de 500 KBits/s.
- El periférico tendrá la capacidad de autosincronizarse y autoresetearse en función de los errores y circunstancias anómalas que puedan tener lugar en el bus de comunicaciones.

### 5.1.3. Requisitos de diseño

Debe ser diseñado e implementado siguiendo los estándares del lenguaje VHDL.

### 5.1.4. Atributos del periférico y su código VHDL

- Código sencillo de mantener y con posibilidades de ampliación.
- Código implementado siguiendo una descripción algorítmica.
- Adaptable a todo tipo de sistemas implementables en FPGAs.
- Robustez en su funcionamiento.

## 5.2. Sistema embebido con procesador Picoblaze

### 5.2.1. Requisitos de interfaces externas

- El sistema embebido debe disponer de una UART para las comunicaciones con el exterior.

### 5.2.2. Requisitos funcionales

- El sistema embebido debe disponer de un microprocesador Picoblaze que controle todos los periféricos del sistema y ejecute el software de control escrito en ensamblador.

### 5.2.3. Requisitos de diseño

Debe diseñarse haciendo uso de IP libres de Xilinx.

## 5.3. Sistema embebido con procesador Microblaze

### 5.3.1. Requisitos de interfaces externas

- El sistema embebido debe disponer de una UART para las comunicaciones con el exterior.
- El sistema embebido debe disponer de una pantalla LCD y leds para mostrar datos.
- El sistema embebido debe disponer de interruptores para cambiar los modos de funcionamiento.

### 5.3.2. Requisitos funcionales

- El sistema embebido debe disponer de un microprocesador Microblaze que controle todos los periféricos del sistema y ejecute el software de control.

### 5.3.3. Requisitos de diseño

Debe diseñarse haciendo uso de IP libres y propietarias(como el propio Microblaze) de Xilinx.

## 5.4. Software de control del sistema embebido

### 5.4.1. Requisitos de interfaces externas

- El programa debe ser capaz de comunicarse con una aplicación externa de monitorización siguiendo un protocolo de comunicaciones diseñado específicamente para ambos. Se usará comunicación serial mediante el protocolo RS-232.

- El sistema debe ser capaz de mostrar información relevante de las tramas recibidas al usuario por una pantalla LCD de tal forma que sea independiente y no necesite de un PC para su funcionamiento.
- El código será específico para el microprocesador Microblaze de Xilinx y sus periféricos, pero será fácilmente modificable para adaptarlo a sistemas de otros fabricantes como Altera, o incluso microprocesadores libres.
- La interfaz de usuario consistirá en una pantalla LCD para mostrar información sobre las tramas recibidas, leds para indicar el funcionamiento y estado del sistema e interruptores para seleccionar los modos de funcionamiento.

### 5.4.2. Requisitos funcionales

- El sistema debe ser capaz de controlar el periférico receptor de tramas.
- El sistema debe ser capaz de extraer información de los registros del periférico receptor.
- El sistema debe ser capaz de mostrar estos datos en la interfaz de usuario.
- El sistema debe poder enviar los datos de la tramas al exterior usando una UART.

### 5.4.3. Requisitos de diseño

Se debe mantener un equilibrio entre el rendimiento que debe ser el máximo posible y el espacio que ocupa debido a los exigentes requisitos de memoria del sistema embebido.

### 5.4.4. Atributos del sistema software

- Código sencillo de mantener y con posibilidades de ampliación.
- Portabilidad entre sistemas embebidos con componentes de distintos fabricantes.
- Máximo rendimiento y robustez en la ejecución.

## 5.5. Software de monitorización del sistema embebido

### 5.5.1. Requisitos de interfaces externas

- El programa debe ser capaz de comunicarse con el sistema embebido anteriormente especificado siguiendo un protocolo de comunicaciones diseñado específicamente para ambos. Se usará comunicación serial mediante el protocolo RS-232.
- El sistema debe ser capaz de mostrar información relevante de las tramas recibidas al usuario.
- El sistema será portable entre los sistemas operativos Windows y GNU/Linux.

- La interfaz de usuario permitirá insertar cinco tramas distintas. Cada trama será identificada por su identificador, identificador extendido(si procede)y descripción. Además se dispondrá de elementos visualizadores para los datos de las tramas recibidas.
- El sistema será capaz de crear un log de la información recibida. Para ello la interfaz debe permitir al usuario seleccionar un fichero donde almacenar el log.
- El formato del log puede ser consultado en el capítulo del desarrollo de la aplicación de monitorización, sección de implementación.

### 5.5.2. Requisitos funcionales

- El sistema debe ser capaz de extraer los datos más relevantes de las tramas recibidas del sistema embebido.
- El sistema debe ser capaz de mostrar la información en la interfaz de usuario en el visualizador correspondiente a la trama a la que pertenecen los datos.
- El sistema debe poder crear un log con los datos recibidos si el usuario selecciona esa funcionalidad.

### 5.5.3. Requisitos de diseño

La rutina de comunicación con el sistema embebido debe ser ejecutada en paralelo a la interfaz gráfica para obtener el máximo rendimiento y no bloquear la interfaz. Para ello será necesario usar hilos.

### 5.5.4. Atributos del sistema software

- Código sencillo de mantener y con posibilidades de ampliación.
- Portabilidad entre sistemas Windows y GNU/Linux.
- Máximo rendimiento y robustez en la ejecución.
- Interfaz amigable al usuario.

# Capítulo 6

# Desarrollo del periférico receptor de tramas CAN

## 6.1. Introducción

En este capítulo se va describir la estructura del receptor de tramas CAN escrito en el lenguaje de descripción VHDL. A este receptor se le denominará periférico debido a que posteriormente será conectado al procesador sofware Microblaze de Xilinx a través de sus buses. Además también será conectado a otro tipo de procesadores embebidos en las FPGA como pueden ser el Picoblaze de Xilinx u otros procesadores desarrollados por la comunidad y que podemos encontrar en OpenCores.

## 6.2. Fase de análisis

En esta fase vamos a analizar los requisitos que tenemos que cumplir a la hora de diseñar el periférico receptor de tramas CAN tanto a nivel del protocolo CAN como de los propios recursos de las FPGA.

Analizaremos primero qué elementos necesitamos para construir un receptor CAN y para ello nos basaremos directamente en el protocolo. A continuación proporcionamos una lista de los elementos que va a tener nuestro periférico que posteriormente comentaremos en sus secciones correspondientes.

- Generador de reloj.
- Registro serie.
- Registro paralelo.
- Máquina de Moore.
- Detector de flanco.
- Calculador CRC.
- Controlador de bit stuffing.

### 6.2.1. Generador de reloj

Según el estándar del protocolo CAN [1], éste puede trabajar a diversas velocidades que oscilan entre los 125 Kbit/s y 1 Mbit/s. Debido a que la placa de evaluación que estamos usando para el desarrollo de este proyecto (Xilinx Spartan 3E Starter Kit) dispone de un oscilador de $50Mhz$ es necesario dividir esta frecuencia puesto que es mucho mayor que la necesaria para alcanzar la velocidad del protocolo CAN.

Nuestro periférico inicialmente estará diseñado para funcionar con el protocolo CAN a la velocidad de 500 Kbit/s [2] por lo que los siguientes cálculos explican qué frecuencia es la necesaria para que nuestro periférico funcione a esa velocidad.

Queremos recibir 500000 bits en un segundo. Si dividimos 1 segundo entre 500000 bits obtenemos que cada $2\mu s$ debe recibirse un bit. Esos $2\mu s$ son el periodo de la frecuencia que necesitamos. Si hacemos la inversa del periodo obtendremos la frecuencia de trabajo el periférico. $1/0,000002\mu s = 500000hz = 500Khz = 0,5Mhz$.

Una vez tenemos calculada la frecuencia de trabajo del periférico, tenemos que conseguirla a partir de los $50Mhz$ del oscilador de la placa de evaluación. Es trivial que dividiendo esa frecuencia entre cien obtenemos la que necesitamos.

El objetivo principal de este bloque es generar una señal de reloj de $0,5Mhz$ a partir de la frecuencia del oscilador de la placa de evaluación de $50Mhz$ para que sea utilizada por el receptor de tramas CAN.

### 6.2.2. Detector de flanco

CAN es un protocolo de comunicación asíncrono, lo que quiere decir que no se envía ninguna señal de reloj junto a los datos. Es más, cada nodo dentro de una red CAN tiene su propio reloj. Si no se envía ninguna señal de reloj, ¿cómo sabemos cuando está llegando una trama nueva?

Puesto que la señal del bus CAN es diferencial, debe ser transformada en una señal única con los niveles de voltaje adecuados para la FPGA (LVTTL)[3]. Esta señal tomará el valor cero lógico cuando la señal señal sea dominante en el bus CAN y uno lógico cuando tome el valor recesivo.

El protocolo CAN lo indica con un bit de inicio de trama que es dominante, es decir, un cero lógico. Puesto que el bus cuando está en reposo se mantiene a recesivo, es decir, uno lógico, se ve claramente que en el momento en el que haya un flanco de bajada comienza una nueva trama.

Para asegurarnos de que es el flanco correcto y no uno situado en el transcurso de la recepción de la trama, el detector de flanco valida el flanco sólo si antes se han recibido diez bits recesivos consecutivos que son los que corresponderían a los siete de final de la trama anterior y los tres de espacio entre tramas como establece el protocolo CAN. Estos diez bits a recesivo no pue-

[1] Estándares ISO 11898-1, ISO 11898-2 y ISO 11898-3
[2] Estándar ISO 11898-5:2007
[3] Low Voltage Transistor-Transistor Logic

den darse nunca en medio de una trama puesto que el mecanismo de bit stuffing no lo permitiría.

Por tanto, el objetivo de este bloque será el de detectar los flancos válidos de inicio de trama y enviar la señal para que comience la captura.

En la siguiente imagen podemos ver como en las tramas, el primer bit (SOF)[4] siempre es dominante e indica el inicio de la trama.

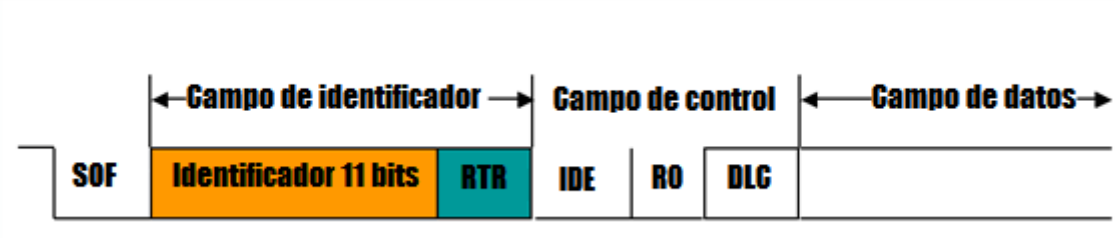


Figura 6.1: Bit de inicio de trama(Autor: Softing)

### 6.2.3. Receptor

Este es el bloque más importante y complejo del periférico. Está compuesto por los siguientes elementos:

#### Registro serie

Como se vio en la sección de definición de las tecnologías, CAN es un protocolo de comunicación serie, lo que quiere decir que los bits se envían secuencialmente por la línea de transmisión. Se hace necesario contar con una serie de registros serie para ir recibiendo la información y que una vez se haya terminado de recibir poder descargar paralelamente hacia otros registros que serán los que conformen la interfaz del periférico con el exterior.

Analizando una trama CAN vemos una serie de campos que son interesantes de almacenar y para ello usaremos registros serie adaptados a su longitud:

- Identificador (CAN tipo A): Registro serie de 11 bits.
- Extensión del identificador (CAN tipo B): Registro serie de 18 bits.
- Número de bytes del campo de datos: Registro serie de 4 bits.
- Campo de datos de la trama: Dos registros serie de 32 bits.
- Campo CRC: Registro serie de 15 bits.

[4]Start of frame

### Registro paralelo

Los registros paralelos que contiene este diseño son la interfaz del periférico con el exterior. Accediendo a estos registros podremos obtener cada uno de los campos de la trama CAN recibida.

A continuación podemos ver un ejemplo de un esquemático de un registro serie-paralelo de 5 bits cuya especificación sería totalmente extensible a las distintas longitudes de los campos que tiene cada trama.

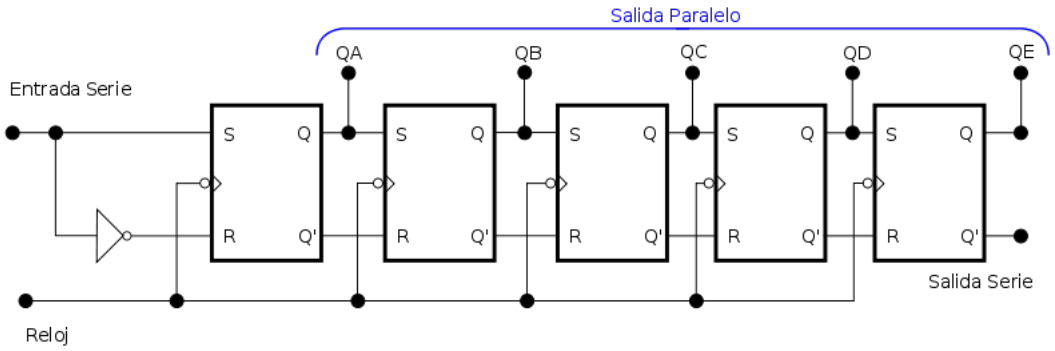


Figura 6.2: Registro serie-paralelo

### Calculador CRC

El protocolo CAN usa el método del CRC para el control de errores en las transmisiones. Por ello es necesario implementar un bloque en el receptor que calcule el CRC de la trama recibida y lo compare con el que se manda en la propia trama. De este modo evitamos almacenar tramas defectuosas y podremos realizar determinadas acciones como resetear el periférico automáticamente si se reciben demasiadas tramas defectuosas.

El polinomio CRC que usa el protocolo CAN es el siguiente:

$$x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$$

### Controlador de bit stuffing

El protocolo CAN usa bit stuffing cuando envía una trama. Por ello el receptor debe estar preparado para detectar estos bits y desecharlos. También debe detectar los posibles errores de bit stuffing que se produzcan y operar pertinentemente.

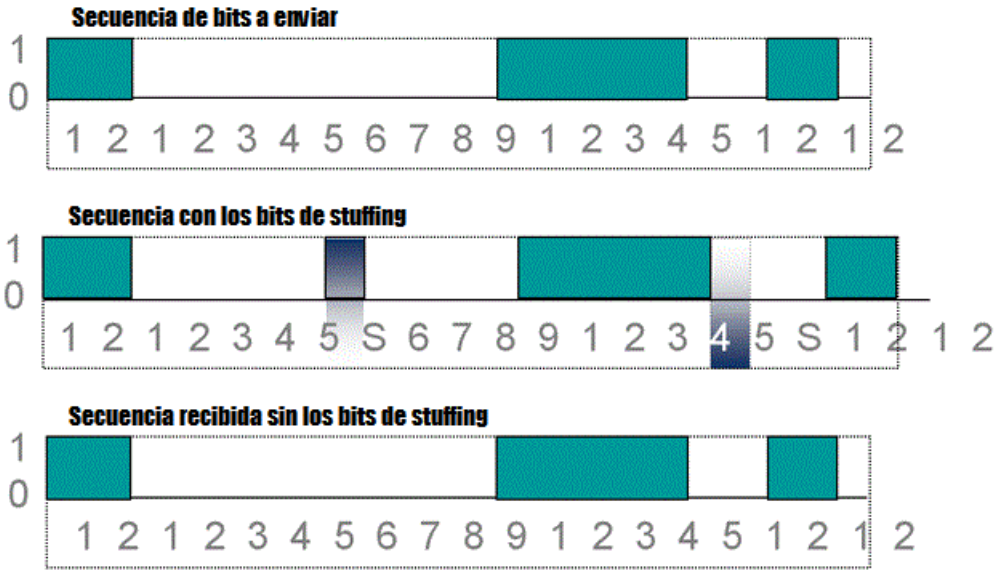


Figura 6.3: Ejemplo de Bit Stuffing

### 6.2.4. Máquina de estados

Una vez vistos todos los bloques anteriores es necesario disponer de uno que controle a todos. La manera más sencilla de controlar los bloques es usando una Máquina de estados de tipo Moore que controle la activación y la sincronización de todos los bloques del periférico.

Las salidas de la máquina de estados que se mostrarán en la tabla de estados son las siguientes (de izquierda a derecha):

- Activar generador de reloj.
- Activar receptor.
- Activar detector de flanco.
- Notificación de error de bit stuffing.
- Notificación de error de CRC.
- Notificación de nueva trama.

| Estado | Salida | Descripción |
|---|---|---|
| S0 | 000000 | Estado inicial. Todos los bloques desactivados. |
| S1 | 001000 | Esperando una nueva trama. |
| S2 | 111000 | Recibiendo una trama. |
| S3 | 000010 | Error de bit stuffing detectado. |
| S4 | 001001 | Error de CRC detectado. |
| S5 | 001010 | Trama recibida con éxito. |

Tabla 6.1: Tabla de estados de la máquina de estados y salidas

Las entradas de la máquina de estados que se mostrarán en la tabla de transiciones son las siguientes (de izquierda a derecha):

- Final de trama.
- Comienzo de trama.
- Error de bit stuffing.
- Error de CRC.

| Entradas | Estado actual | Siguiente estado |
|---|---|---|
| - - - - | S0 | S1 |
| - 0 - - | S1 | S1 |
| - 1 - - | S1 | S2 |
| - - 1 - | S2 | S3 |
| - - 0 1 | S2 | S4 |
| 0 - 0 0 | S2 | S2 |
| 1 - 0 0 | S2 | S5 |
| - - - - | S3 | S1 |
| - - - - | S4 | S1 |
| - - - - | S5 | S1 |

Tabla 6.2: Tabla de transición de estados de la máquina de estados

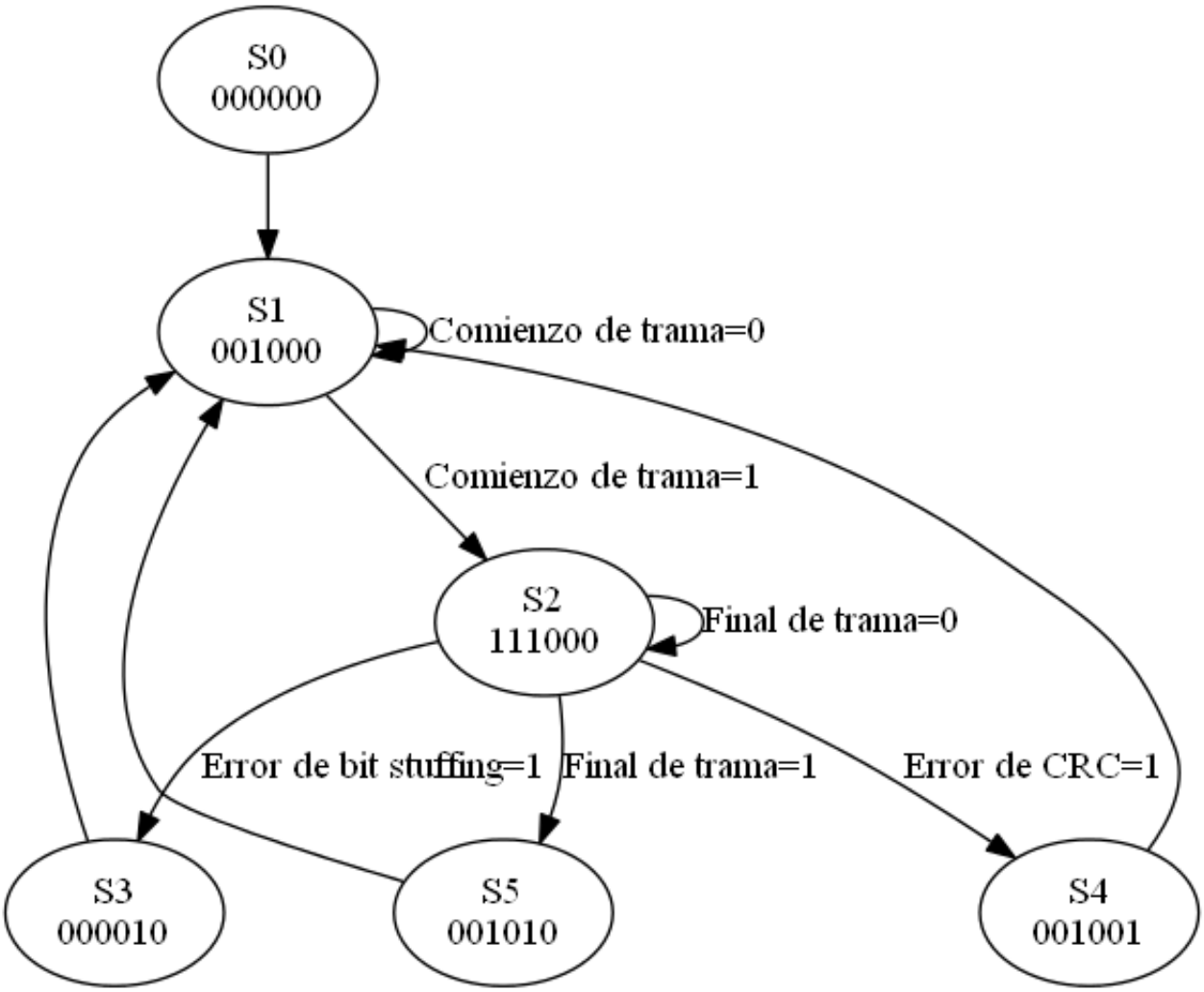


Figura 6.4: Grafo de la máquina de estados

## 6.3. Fase de diseño

Una vez realizado el análisis de todos los requisitos necesarios para el desarrollo de un periférico de lectura de tramas CAN, es hora de realizar el diseño del mismo basándonos en todas las especificaciones definidas en la sección anterior.

En esta fase vamos a describir la estructura de cada bloque del periférico definido anteriormente con sus entradas y salidas y su interacción con el resto de los bloques.
Además se describirá la estructura global del periférico mediante la interconexión de todos sus componentes, así como los bancos de registros que podrán ser accedidos desde el exterior para leer del periférico.

En esta fase se introducirán también los diagramas de flujo de cada componente y del periférico completo para observar el comportamiento de cada uno y tener una referencia para luego

describirlos mediante un lenguaje de descripción de hardware en la sección de implementación.

### 6.3.1. Bloque superior del periférico

El periférico presentará las siguientes entradas y salidas:

#### Entradas y salidas

Las entradas y salidas que tiene este bloque y sus descripciones son las siguientes:

| Señal | Tipo | Descripción |
|---|---|---|
| Clk | Entrada | Reloj de funcionamiento del periférico. |
| Reset | Entrada | Señal de reset del bloque |
| Rx | Entrada | Entrada de datos de la transmisión. |
| Data1(31:0) | Salida | Registro D1. |
| Data2(31:0) | Salida | Registro D2. |
| Id(31:0) | Salida | Registro ID. |
| EId(31:0) | Salida | Registro EID. |

Tabla 6.3: Entradas y salidas del **periférico**

#### Diagrama de bloques

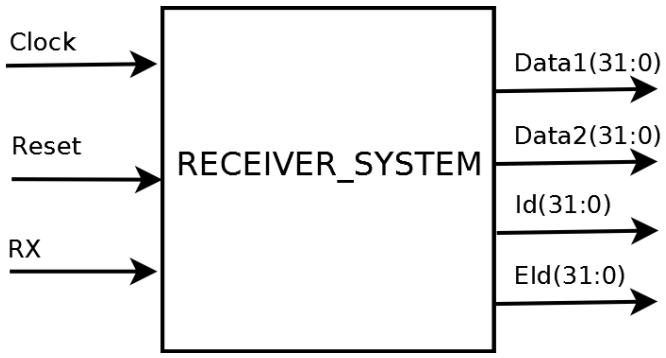


Figura 6.5: Diagrama de bloques del bloque superior del periférico

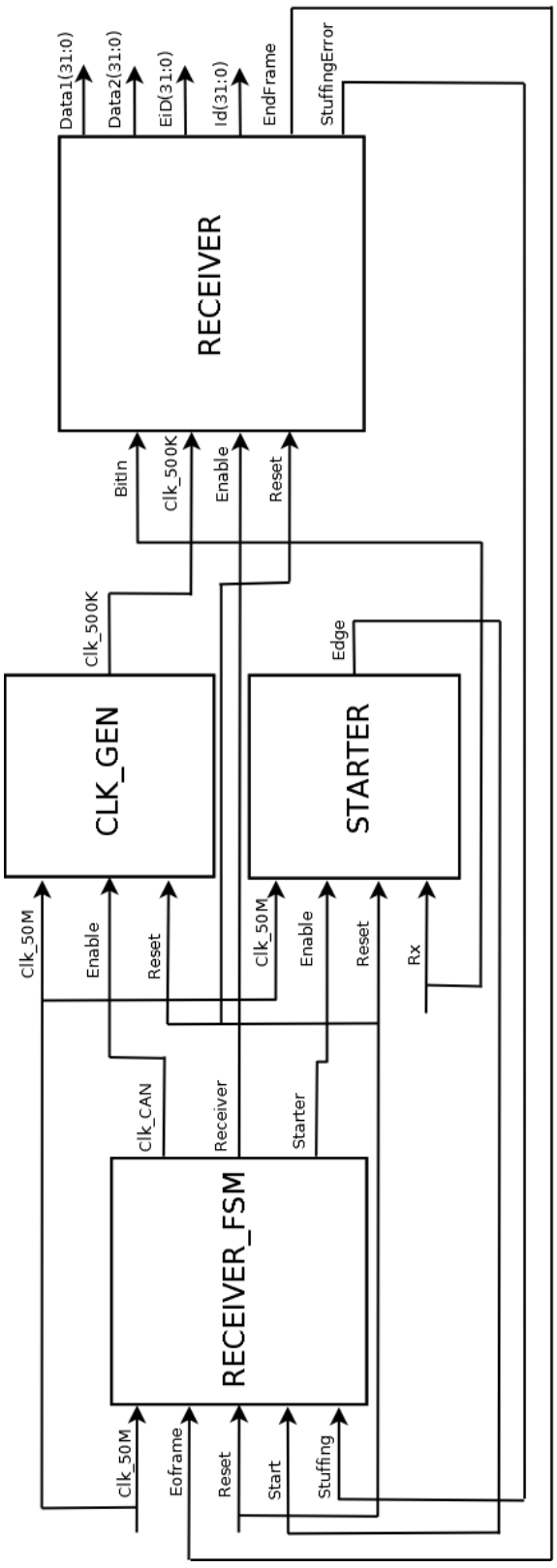


Figura 6.6: Diagrama del conjunto de bloques que constituyen el periférico

**Diagrama de flujo**

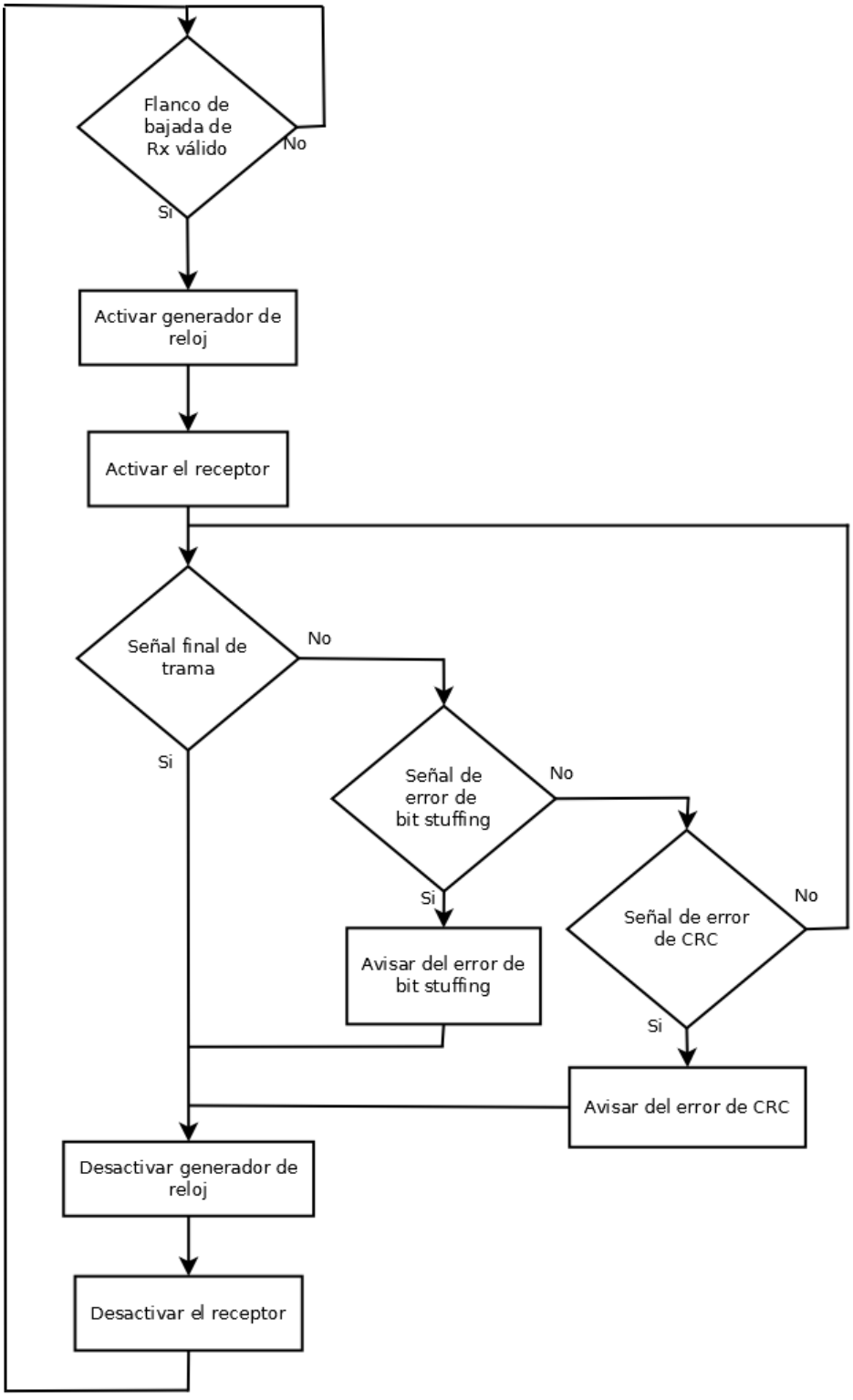


Figura 6.7: Diagrama de flujo del funcionamiento del periférico

A continuación desglosaremos cada bloque que compone el de la figura 4.5:

### 6.3.2. Generador de reloj

Este bloque provee al periférico de la lógica necesaria para adaptar la frecuencia de reloj disponible en la placa de evaluación a la necesaria para recibir las tramas a la velocidad del estándar. Como se comentó en el análisis, la placa de evaluación dispone de un oscilador de 50 MHz, por lo que es necesario dividir los 50 MHz hasta alcanzar 0,5 MHz que la frecuencia de reloj que nos dará la velocidad de 500 Kbits/s, o lo que es lo mismo, 2 $\mu$s de tiempo de bit.

#### Entradas y salidas

Las entradas y salidas que tiene este bloque y sus descripciones son las siguientes:

| Señal | Tipo | Descripción |
|---|---|---|
| Clock | Entrada | Reloj de 50 MHz. |
| Reset | Entrada | Señal de reset del bloque |
| Enable | Entrada | Activación del bloque. |
| Clk_500K | Salida | Señal de reloj de 0,5 MHz producida |

Tabla 6.4: Entradas y salidas del bloque **Generador de reloj**

#### Diagrama de bloques

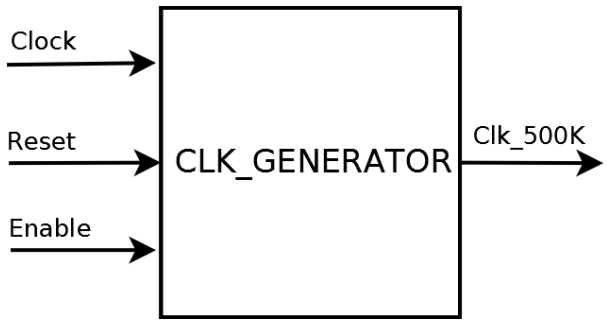


Figura 6.8: Diagrama de bloques del generador de reloj

**Diagrama de flujo**

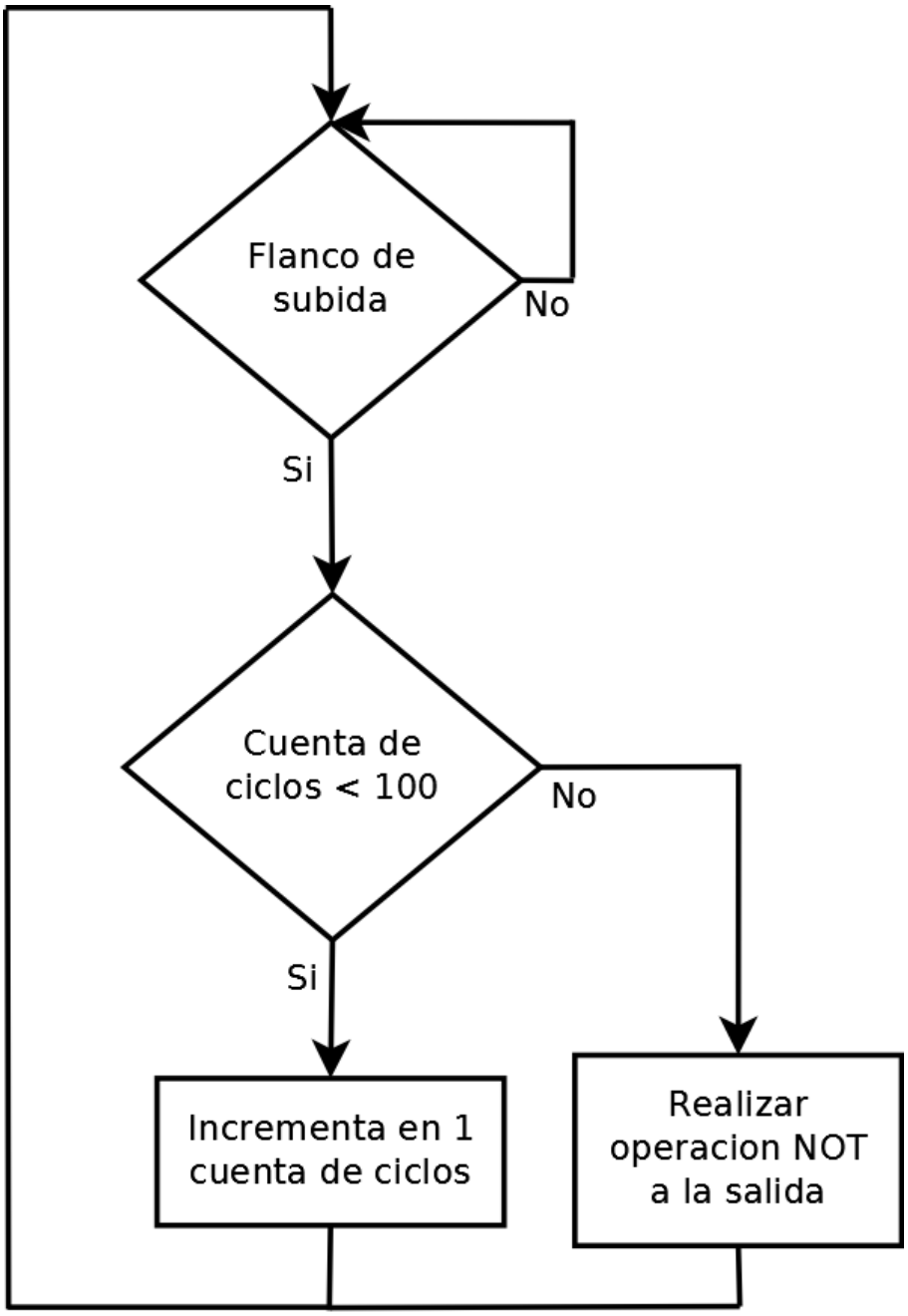


Figura 6.9: Diagrama de flujo del generador de reloj

### 6.3.3. Detector de flanco

Este bloque provee al periférico de la lógica necesaria para detectar flancos de bajada del nivel del bus de transmisión que son los que indican el comienzo de una nueva transmisión. Para que el flanco de bajada sea el que indique el comienzo de la trama y no uno que ocurra en el transcurso de una transmisión, debe estar precedido de 10 bits recesivos, es decir, de valor uno

lógico. Estos 10 bits recesivos se corresponden con los de terminación de la trama (7 bits) y los tres bits de espacio entre tramas y sólo pueden darse al final de las tramas debido a que el mecanismo de “bit stuffing” impide que se de esta circunstancia en mitad de la transmisión de una trama.

El bloque tiene en cuenta lo mencionado anteriormente y una vez haya esperado diez bits a 1 detecta el siguiente flanco de bajada a estos.

### Entradas y salidas

Las entradas y salidas que tiene este bloque y sus descripciones son las siguientes:

| Señal | Tipo | Descripción |
|---|---|---|
| Clock | Entrada | Reloj de 50 MHz. |
| Reset | Entrada | Señal de reset del bloque |
| Enable | Entrada | Activación del bloque. |
| Rx | Entrada | Entrada de la transmisión. Detecta los flancos de bajada de esta señal |
| Start | Salida | Estará activa cuando llegue un flanco válido en la señal Rx. |

Tabla 6.5: Entradas y salidas del bloque **Detector de flanco**

### Diagrama de bloques

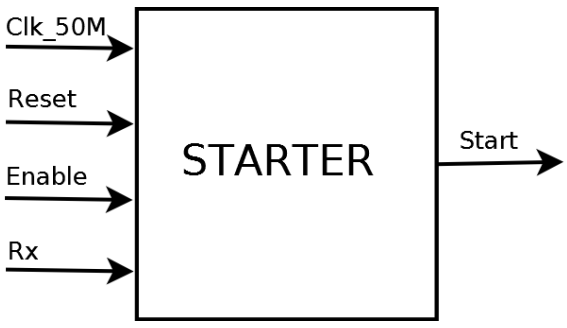


Figura 6.10: Diagrama de bloques del detector de flanco

**Diagrama de flujo**

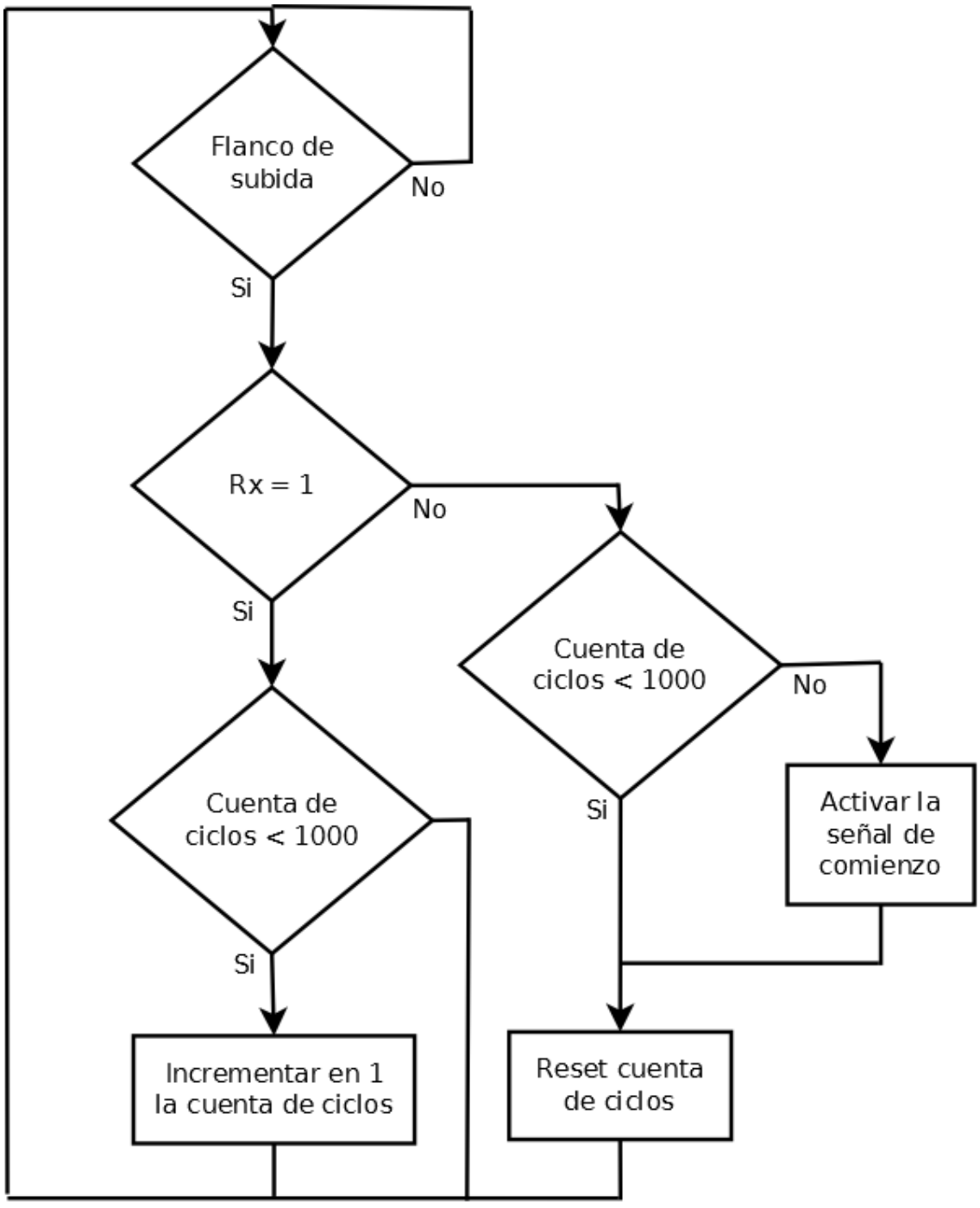


Figura 6.11: Diagrama de flujo del detector de flanco

### 6.3.4. Máquina de estados

Este bloque controla el funcionamiento y sincronización del resto de los bloques que componen el periférico. Se trata de una máquina de Moore cuyas salidas son las señales de activación del

resto de los bloques. Además controla los errores que se puedan producir en la transmisión y resetea el periférico en caso de que tenga lugar uno de estos errores.

### Entradas y salidas

Las entradas y salidas que tiene este bloque y sus descripciones son las siguientes:

| Señal | Tipo | Descripción |
|---|---|---|
| Clk_50M | Entrada | Reloj de 50 MHz. |
| Eoframe | Entrada | Indica se ha finalizado la recepción de una trama. |
| Reset | Entrada | Señal de reset del bloque |
| Start | Entrada | Indica si ha sido detectado un flanco de bajada. |
| Stuffing | Entrada | Indica si se ha producido un error de bit stuffing. |
| CLK_CAN | Salida | Señal de activación del bloque generador de reloj. |
| Receiver | Salida | Señal de activación del bloque receptor de trama. |
| Starter | Salida | Señal de activación del bloque detector de flancos. |

Tabla 6.6: Entradas y salidas del bloque **Máquina de estados**

### Diagrama de bloques

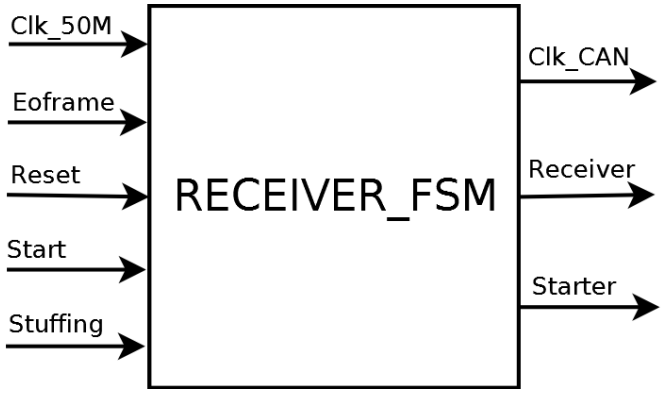


Figura 6.12: Diagrama de bloques de la máquina de estados

**Diagrama de flujo**

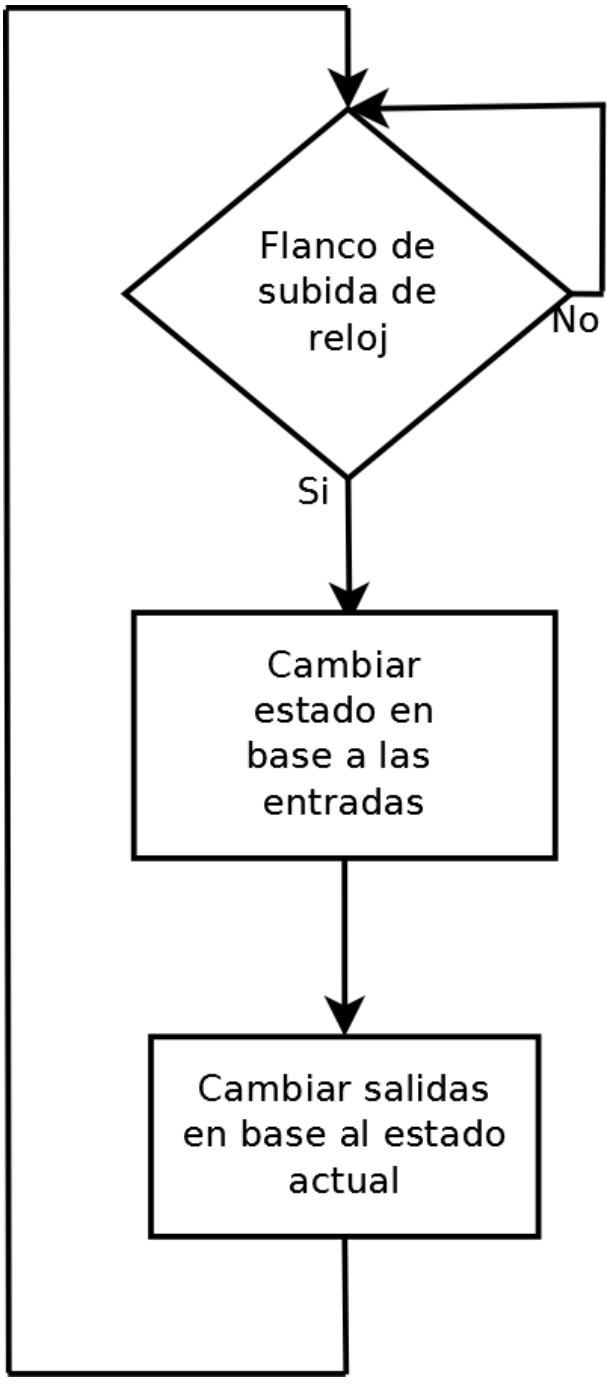


Figura 6.13: Diagrama de flujo de la máquina de estados

### 6.3.5. Receptor

Este bloque controla la recepción de las tramas CAN. Dentro de él se lleva a cabo el control del mecanismo del Bit-Stuffing, el cálculo del CRC-15 y su posterior validación y se recibe la

trama dividiéndola en campos que posteriormente se almacenarán en registros de 32 bits que se convertirán en la interfaz del periférico con el exterior.

### Entradas y salidas

Las entradas y salidas que tiene este bloque y sus descripciones son las siguientes:

| Señal | Tipo | Descripción |
|---|---|---|
| BitIn | Entrada | Entrada de datos de la transmisión. |
| CLK_500K | Entrada | Reloj de funcionamiento del receptor. |
| Reset | Entrada | Señal de reset del bloque |
| Enable | Entrada | Señal de activación del bloque. |
| EndFrame | Salida | Se activa cuando la recepción de la trama ha llegado a su fin. |
| StuffinError | Salida | Se activa si ha tenido lugar un error de bit stuffing. |
| Data1(31:0) | Salida | Registro 1 del campo de datos. |
| Data2(31:0) | Salida | Registro 2 del campo de datos. |
| Id(31:0) | Salida | Registro del identificador. |
| EId(31:0) | Salida | Registro del identificador extendido. |

Tabla 6.7: Entradas y salidas del bloque **Receptor**

### Diagrama de bloques

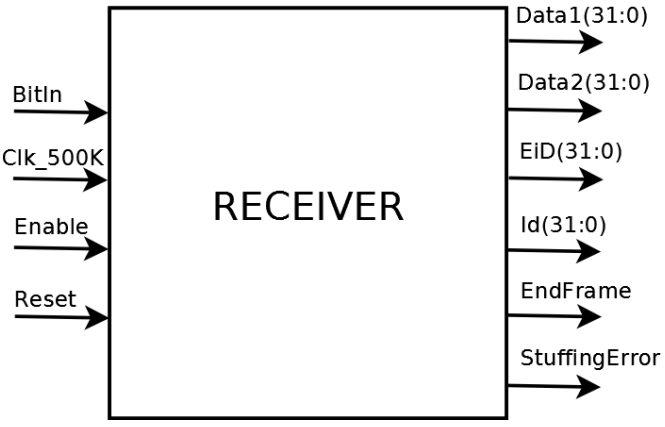


Figura 6.14: Diagrama de bloques del receptor

**Diagrama de flujo**

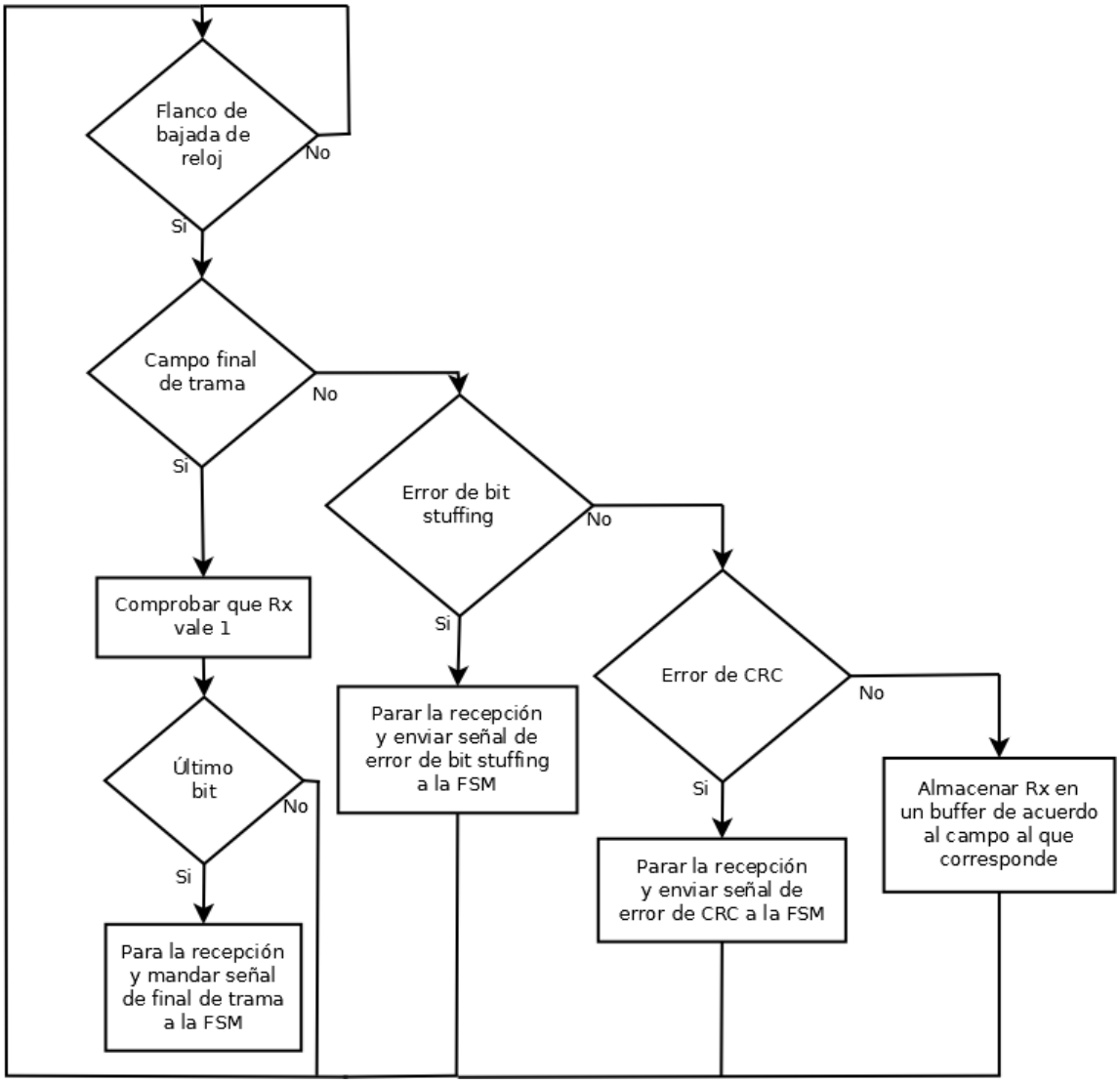


Figura 6.15: Diagrama de flujo del receptor

### 6.3.6. Interfaz con el exterior

La interfaz con el exterior de este periférico consistirá en un banco de cuatro registros en los que se almacenará toda la información importante que se extraiga de cada trama. Esta información será la que se muestra en la siguiente tabla:

| Dato | Longitud (Bits) |
|---|---|
| Identificador de la trama | 11 |
| Identificador extendido de la trama | 18 |
| Bit de trama remota | 1 |
| Bit de trama con identificador extendido | 1 |
| Longitud del campo de datos | 4 |
| Datos | 0-64 |
| CRC | 15 |

Tabla 6.8: Datos extraíbles de una trama

### Organización de los registros

Estos datos hay que agruparlos en cuatro registros. El criterio que se ha seguido a la hora de elegir la longitud en bits de estos registros ha sido la longitud de palabra de los procesadores existentes. Teniendo en cuenta que la mayoría de "soft-cores"de hoy día son de 32 bits, es obvio que la longitud de los registros debe ser también de 32 bits. Con esta elección no perjudicamos ni a los procesadores de 32 y 64 bits que podrán leer los registros en una única operación de lectura, ni a los procesadores de 16 bits o de 8 bits que realizarán dos y cuatro lecturas respectivamente y que no hay posibilidad de reducir cambiando la palabra de los registros.

La organización de los datos extraídos de una trama en los registros se muestra en las siguientes tablas. Cuando un programa software quiera obtener un determinado campo de la trama recibida no tiene más que leer el correspondiente registro donde se encuentre el campo deseado y almacenar los bits correspondientes en una variable.

### Registro ID

| Bit | Símbolo | Descripción | Valor de reset |
|---|---|---|---|
| 10:0 | ID | Identificador de la trama | 0 |
| 11 | - | Reservado. No usar | 0 |
| 26:12 | CRC | CRC de la trama | 0 |
| 27 | - | Reservado. No usar | 0 |
| 31:28 | DLC | Longitud del campo de datos | 0 |

Tabla 6.9: Campos del registro ID

**Registro EID**

| Bit | Símbolo | Descripción | Valor de reset |
|---|---|---|---|
| 17:0 | EID | Identificador extendido de la trama | 0 |
| 27:18 | - | Reservado. No usar | 0 |
| 28 | CRCE | Error CRC | 0 |
| 29 | STUE | Error de bit stuffing | 0 |
| 30 | NFRA | Nueva trama | 0 |
| 31 | EFR | Trama con identificador extendido | 0 |

Tabla 6.10: Campos del registro EID

**Registro D1**

| Bit | Símbolo | Descripción | Valor de reset |
|---|---|---|---|
| 31:0 | B1-4 | Bytes 1 a 4 de datos de la trama | 0 |

Tabla 6.11: Campos del registro D1

**Registro D2**

| Bit | Símbolo | Descripción | Valor de reset |
|---|---|---|---|
| 31:0 | B5-8 | Bytes 5 a 8 de datos de la trama | 0 |

Tabla 6.12: Campos del registro D2

## 6.4. Fase de implementación

Esta es la fase en la que se va a codificar el periférico usando el lenguaje de descripción de hardware VHDL y siguiendo todos los requisitos que se establecieron en las fases de análisis y diseño.

A continuación se expondrán las técnicas utilizadas para codificar en VHDL tanto usadas en todos los bloques como individualmente en cada uno. Posteriormente se expondrá como se ha implementado cada bloque individualmente.

### 6.4.1. Técnicas generales de codificación en VHDL

En esta subsección se explicará que elementos provee el lenguaje VHDL para tratar los distintos elementos presentes en el hardware como pueden ser los registros, la señales de reloj,

contadores, la señal de reset, etc...

Un aspecto a tener en cuenta cuando se codifica en VHDL es modularizar el código en cada uno de los bloques que van a componer nuestro periférico, es decir, codificar cada bloque en ficheros VHDL individuales. Con esto aseguramos mayor legibilidad al código y facilidad en la depuración.

Pasamos a describir la estructura de un módulo VHDL.

### Estructura de un módulo VHDL

Un módulo VHDL se compone de varias secciones bien diferenciadas:

- Declaración de bibliotecas: Debe situarse al principio de cada módulo.
- Entity: Define cuales van a ser la señales de entrada y salida de un módulo VHDL.
- Architecture: En ella se define el comportamiento del módulo. Se compone de señales, procesos y sentencias concurrentes.
    - Señales: Declaradas antes del "begin". Pueden ser usadas tanto para entradas como para salidas. El tipo puede ser cualquiera de los que define el estándar y sus bibliotecas aunque los más empleados son "integer", "std_logic" y "std_logic_vector". Los tipos "std_logic" se usan en los módulos que luego van a ser implementados en una fpga real.
    - Proceso: Un proceso es una función que se ejecuta concurrentemente a otros procesos declarados y que es llamada cuando una de las señales de su lista de sensibilidad (variables entre paréntesis) cambia de valor. Se suelen utilizar cuando la acción depende de una señal de reloj.
    - Sentencia concurrente: Se denomina sentencia concurrente a toda aquella que está fuera de un proceso. Se utilizan mucho para asignar valores a las salidas ya que a diferencia de los procesos estas sentencias se van a ejecutar siempre.

A continuación podemos ver un pequeño fragmento de código que muestra lo anteriormente comentado:

```
--Esto es un comentario
--Bibliotecas
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--Definicion del bloque
entity EJEMPLO is
    Port( a : in std_logic;
```

```
          b : out std_logic);
end EJEMPLO;

--Arquitectura del bloque
architecture Behavioral of EJEMPLO is

--Constante entera
constant constante: integer := 650;

--Declaracion de senales
signal cuenta: integer range 0 to 9;
signal salida: std_logic;

begin

--Asignacion paralela
b <= a;

--Proceso
process (a,b)
begin
        if a = '1' then
                salida <= '0';
        else
                cuenta <= cuenta +1;
        end if;
end process;

end Behavioral;
```

### Señales de reloj y reset

En VHDL las señales de reloj y de reset deben ser tratadas de forma diferente al resto de las señales. Esto es debido a que las herramientas de síntesis y ruteado intentan, para estas señales, usar rutas específicas que aseguran que las señales tendrán los mínimos retardos posibles y que lleguen con la misma intensidad a todos los puntos donde se necesiten.

Es por ello que las señales de reloj y reset del periférico deben tener un nombre como los siguientes:

- Señal de reloj: CLK, clk, clock, clock_nombre, etc...
- Señal de reset: RST, rst, reset, reset_nombre, etc...

Una vez conocemos como declarar este tipo de señales, el siguiente ejemplo de código muestra como usar adecuadamente las señales de reloj y reset.

```
--Declaracion de senales
```

```
signal clk: std_logic;
signal rst: std_logic;

--Proceso
process (clk,rst)
begin
        if rst = '1' then
                --Acciones a realizar durante el reset
        elsif rising_edge(clk) then
                --Acciones a realizar cuando es detectado
                --un flanco de subida
        end if;
end process;
```

Como podemos ver en el ejemplo, ambas señales son utilizadas dentro de procesos puesto que lo que interesa de este tipo de señales son los momentos de cambio de valor y para eso se incluyen en la lista de sensibilidad de un proceso.
Se utilizan dos funciones para detectar los flancos en la señal de reloj:

- rising_edge: detecta los flancos de subida de la señal de reloj.
- falling_edge: detecta los flancos de bajada de la señal de reloj.

A lo largo de los módulos del periférico aparecerán las señales de reloj y reset como se han especificado en esta sección.

### Registros

A lo largo del periférico se emplearán una serie de registros serie y paralelos que almacenarán tanto información temporal como los datos recibidos de las tramas que posteriormente serán leídos por un procesador.

Los registros se declaran en VHDL como vectores. En el caso del periférico serán de tipo "std_logic_vector". Los vectores sirven tanto como registros paralelos como serie.
En el siguiente ejemplo vemos la forma en la que usarán en el periférico:

```
--Vector que representa el registro.
signal registro: std_logic_vector(7 downto 0);

--Carga en paralelo
registro<="00110011";

--Carga serie hacia la izquierda
registro<=registro(6 downto 0)&'1';

--Carga serie hacia la derecha
registro<='1'&registro(7 downto 1);
```

### 6.4.2. Bloque: Generador de reloj

El bloque generador de reloj se compone de un único proceso. Este proceso es el que detecta los flancos de subida del reloj de 50 Mhz y a cada flanco incrementa un contador. Como se explicó en la fase de análisis, es necesario dividir entre cien la señal de 50 Mhz para obtener los 500 Khz que necesitamos para los 500 Kbits de velocidad que definimos en el análisis de los requisitos. Para dividir la señal usamos ese contador.

Ese contador será una señal de tipo "integer" con rango de 0 a 49 que es justo la mitad del periodo de reloj. Cuando se alcanza el límite del contador, se cambia la polaridad de la señal de salida. De esta forma se genera una señal de reloj con un periodo cien veces menor.

### 6.4.3. Bloque: Detector de flanco

El bloque detector de flanco se compone de un único proceso. Este proceso es el que detecta los flancos de subida del reloj de 50 MHz y a cada flanco incrementa un contador si la señal Rx está a uno lógico.El contador tiene un rango de 0 a 1000. El contador llega hasta 1000 puesto que es el número de ciclos que corresponden a diez bits según la velocidad establecida en la fase de análisis. Si en algún momento de la cuenta la señal Rx cambia a cero lógico se resetea la cuenta. Esta cuenta se corresponde con los diez bits a uno lógico que debe haber entre dos tramas.

Si el contador llega con éxito a 1000 significa que han llegado 10 bits a uno lógico y que por tanto el siguiente flanco de bajada será válido. Una vez llegado a este punto, el detector de flanco esperará a que llegue un flanco de bajada en la señal Rx para dar la señal de comienzo a la maquina de estados y que ésta inicie el proceso de recepción de la trama.

### 6.4.4. Bloque: Maquina de estados

La máquina de estado del periférico se compone de los siguientes tres procesos:

- Un proceso que cambia de estado cada flanco de subida del reloj en base a una variable que contiene el estado siguiente al que cambiar.
- Un proceso que en base al valor de las entradas y del estado actual cambia la variable de estado siguiente al correspondiente.
- Un tercer proceso que en base al estado actual coloca las salidas de la máquina de estado al valor que se especificó en la fase de diseño.

### 6.4.5. Bloque: Receptor

El bloque correspondiente al receptor de tramas es con diferencia el más complejo de todos los que componen el periférico. Además es el que ha necesitado más tiempo para poder codificar en VHDL su especificación de diseño. El código VHDL se compone de lo siguiente:

- 6 registros serie-paralelo para almacenar cada uno de los campos de la trama.
- 4 registros paralelos que se corresponden con los registros del periférico especificados en el diseño.
- 9 contadores para llevar el control de los bits de cada campo que van llegando.
- 11 señales de control para alternar entre procesos.
- 10 señales para el control del bit stuffing y el CRC.
- 12 procesos.

Como los procesos son la parte más importante del código VHDL, es importante conocer cual es la función de cada uno y el orden en el que se ejecutan. Todos los procesos del receptor están etiquetados según su función. En la siguiente tabla queda recogida toda esa información:

| Proceso | Descripción |
|---|---|
| IDENTIFIER | Recibe el identificador de 11 bits y lo almacena en un registro serie temporal. |
| CONTROL | Recibe los bits de control de la trama. |
| DATALENGTH | Recibe la longitud del campo de datos de la trama y lo almacena en un registro serie temporal. |
| EXTENDEDID | Recibe el identificador extendido de 18 bits y lo almacena en un registro serie |
| DATAFIELD | Recibe el campo de datos de la trama y lo almacena en dos registros temporales. |
| CRCFIELD | Recibe el campo CRC de la trama y lo almacena en un registro serie temporal. |
| ENDOFFRAME | Comprueba que los bits de final de trama llegan correctamente. |
| SAVE | Descarga los registros serie de los campos de la trama en los registros paralelos. |
| STUFF | Controla el mecanismo de bit stuffing. |
| BITSTUFF | Desactiva el resto de procesos cuando llega un bit de stuffing. |
| CRC | Calcula el CRC de la trama que se está recibiendo al vuelo. |
| CRC_CHECK | Comprueba que el CRC calculado al vuelo es idéntico al recibido en la trama. |

Tabla 6.13: Procesos VHDL del bloque receptor de tramas

## 6.5. Fase de pruebas

En esta fase se verifica que todos los bloques que se implementaron en la fase anterior tienen el comportamiento que se especificó en la fase de diseño. Para realizar las pruebas se ha utilizado un simulador de VHDL. Usando este simulador(*Isim*) se comprobaron si las salidas de cada bloque eran las esperadas en base a las entradas.

A continuación se exponen los cronogramas de cada bloque donde se puede observar su correcto comportamiento.

### 6.5.1. Bloque: Generador de reloj

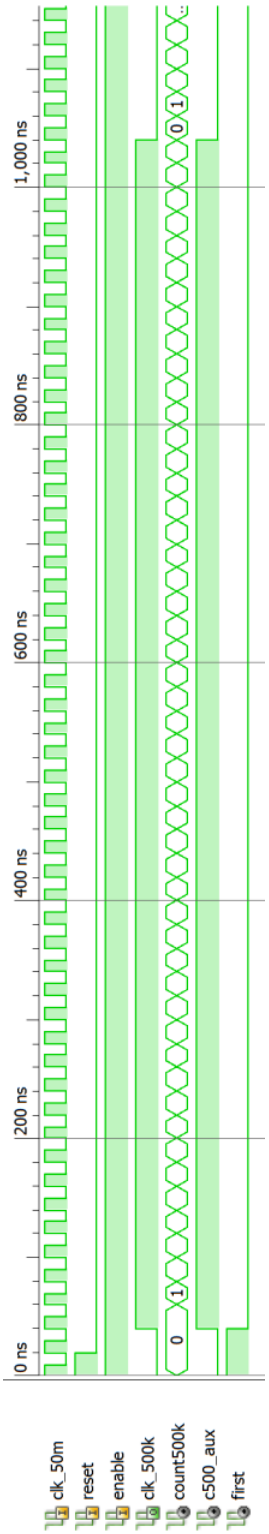


Figura 6.16: Cronograma del generador de reloj

### 6.5.2. Bloque: Detector de flanco

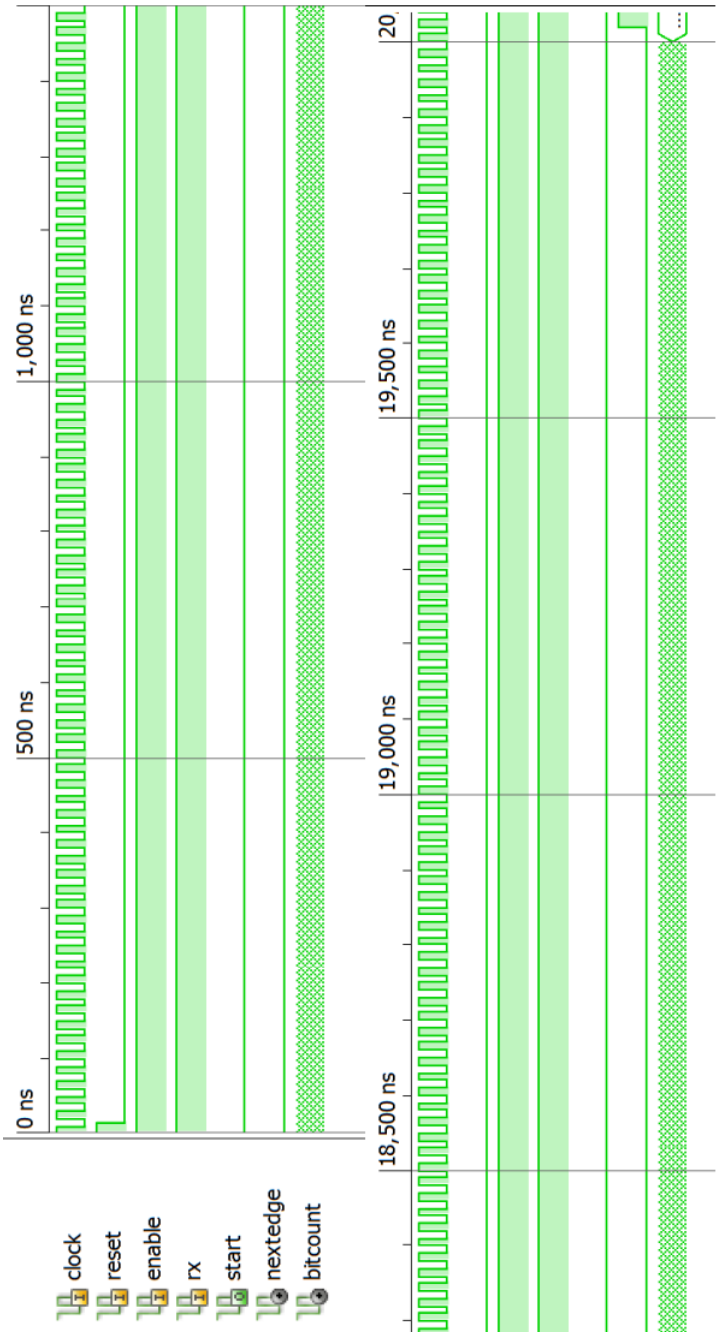


Figura 6.17: Cronograma del detector de flanco

### 6.5.3. Bloque: Máquina de estados

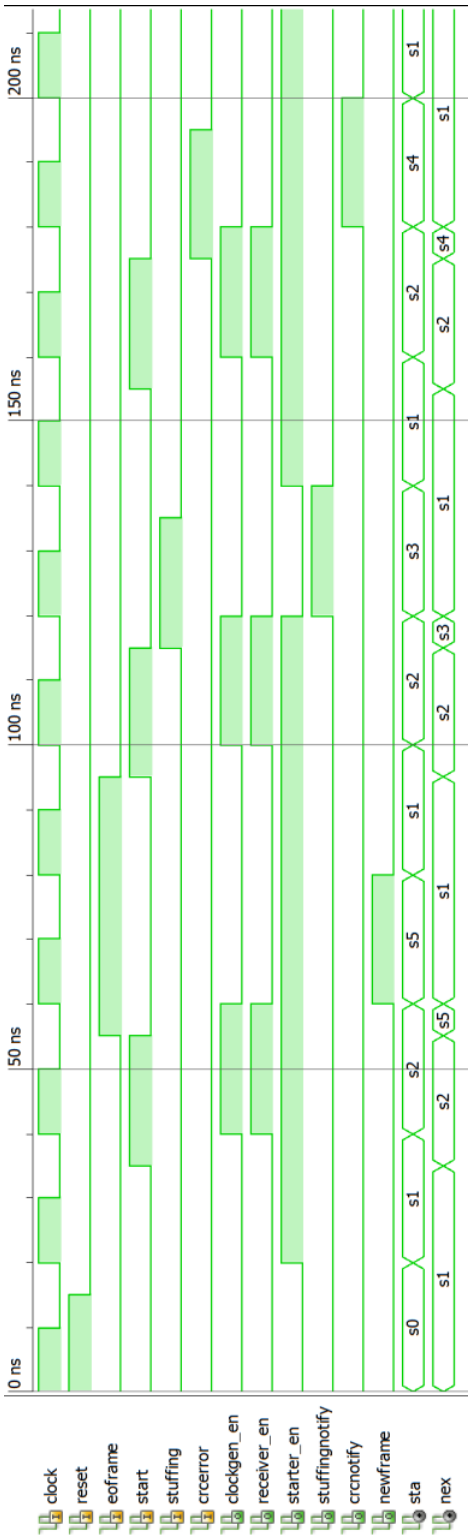


Figura 6.18: Cronograma de la máquina de estados

### 6.5.4. Bloque: Receptor

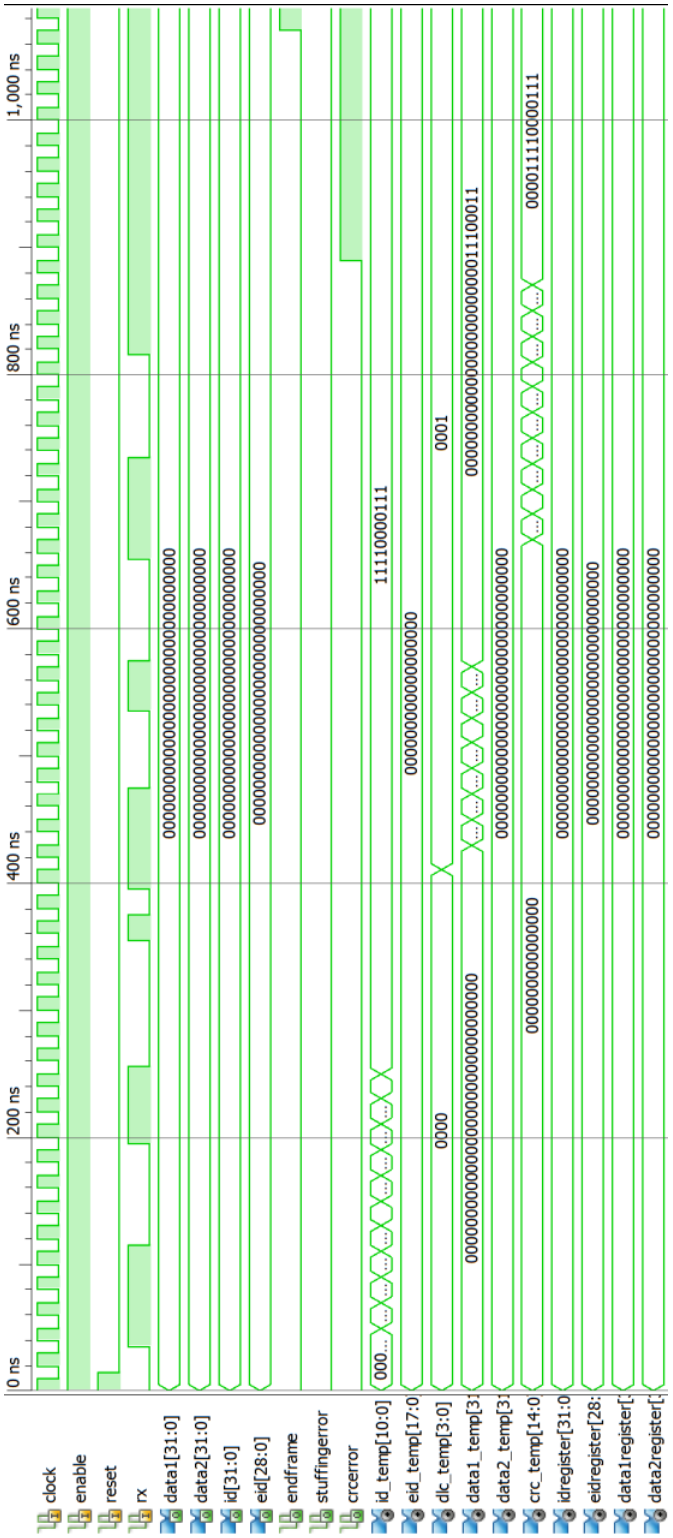


Figura 6.19: Cronograma del receptor 1

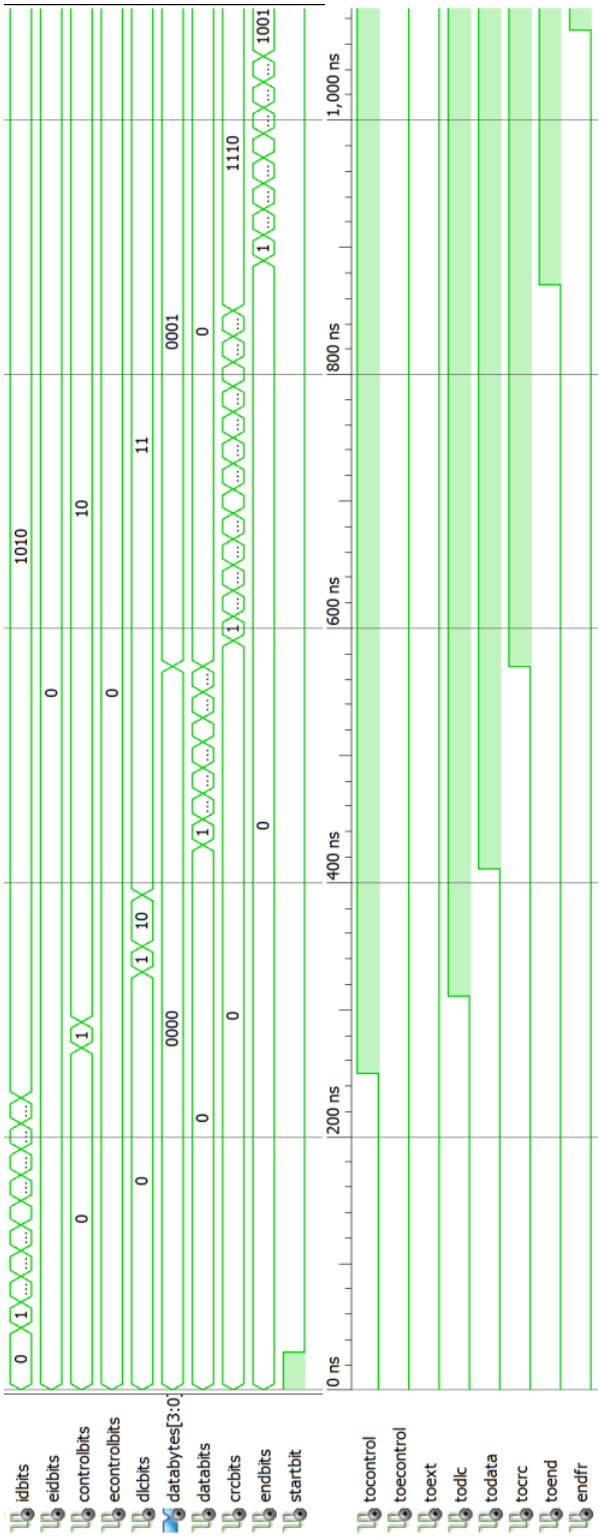


Figura 6.20: Cronograma del receptor 2

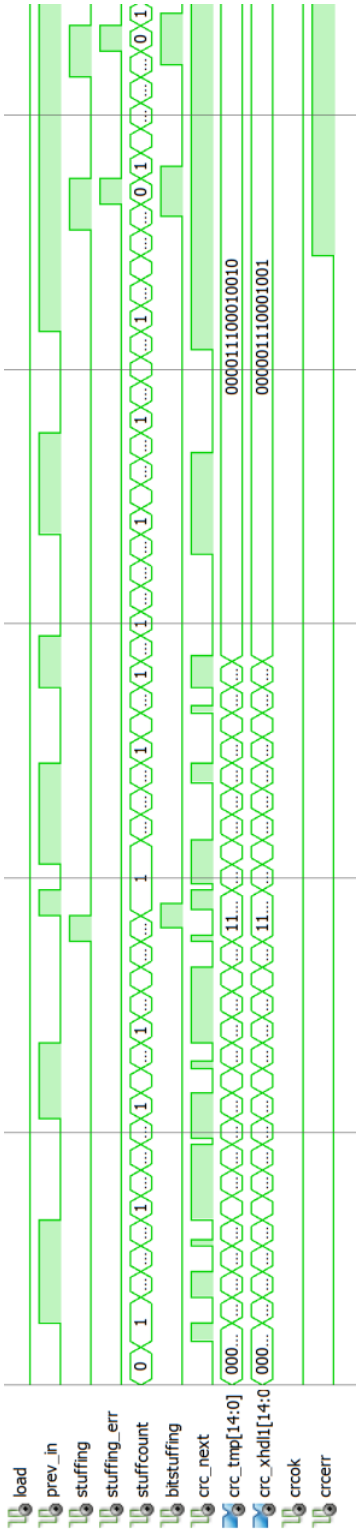


Figura 6.21: Cronograma del receptor 3

Además de simular todos los componentes con *Isim*, también se probó el correcto funcionamiento mediante su implementación en la placa de evaluación Spartan 3E Starter Kit y su conexión a un microcontrolador LPC 2378 con núcleo ARM que posee un periférico CAN y

que se usó para el envío de tramas al periférico que ha sido desarrollado en VHL.

# Capítulo 7

# Desarrollo del sistema embebido con procesador Picoblaze de Xilinx

## 7.1. Introducción

En este capítulo se expondrá todo el proceso de desarrollo que se ha seguido para poner en funcionamiento un sistema embebido basado en el microprocesador software Picoblaze. Este sistema embebido incluirá dos recursos fundamentales entre los que estará el periférico receptor de tramas del capítulo anterior y una UART para las comunicaciones con el exterior.

## 7.2. Fase de análisis

En esta fase vamos a analizar los requisitos que tenemos que cumplir a la hora de diseñar el sistema embebido tanto a nivel del periférico receptor como de los propios recursos que presenta la FPGA.

Analizaremos primero qué elementos necesitamos para diseñar un sistema embebido que tenga la funcionalidad que hemos comentado en la introducción. A continuación proporcionamos una lista de los elementos que va a necesitar nuestro sistema embebido y que posteriormente comentaremos en sus secciones correspondientes. Todos estos componentes serán implementados en una única FPGA.

- Procesador software Picoblaze de Xilinx [1].
- Memoria PROM para almacenar el código que ejecutará el procesador [2].
- UART [3].
- Periférico receptor de tramas CAN (Desarrollado en el capítulo 6 de este proyecto).

[1] IP Core libre ofrecido por Xilinx
[2] Pertenece a la FPGA
[3] IP Core libre ofrecido por Xilinx

### 7.2.1. Procesador software Picoblaze de Xilinx

El procesador software es la parte más importante del sistema embebido ya que sin él no podrían funcionar el resto de los elementos del sistema. Se trata de un procesador que posee las siguientes características:

- Ancho de palabra de 8 bits.
- Tipo RISC.
- Memoria de instrucciones PROM de 1K x 18
- Cada instrucción se ejecuta en dos ciclos de reloj. Es por ello que el rendimiento en MIPS es la mitad de la frecuencia de reloj que recibe. Si la señal de reloj es de 50 MHz, por ejemplo, el Picoblaze dará un rendimiento de 25 MIPS.
- Ocupa 96 Slices de la FPGA y 1 block RAM.
- 16 registros de 8 bits.
- Programación en lenguaje ensamblador.
- Código abierto.

En cuanto a la frecuencia del sistema, la placa de evaluación que se está usando en este proyecto de un cristal oscilador de 50 MHz. Emplearemos esta frecuencia para el Picoblaze.

### 7.2.2. Memoria PROM

En todo sistema embebido es necesario disponer de una determinada cantidad de memoria para almacenar el programa que ejecuta el procesador. En el sistema embebido que se va a implementar en este capítulo, se necesita una pequeña memoria para almacenar el programa.

El código proporcionado por Xilinx para la memoria del procesador se implementa sobre los recursos **"block RAM"** de la FPGA.

Al tratarse de un sistema basado en Picoblaze, se dispondrá de hasta 1K x 18 direcciones de memoria para almacenar el programa ensamblador.

### 7.2.3. UART

Para comunicar el diseño con el exterior será necesario disponer de una UART. La UART[4]presenta los siguientes parámetros a configurar:

- Buffer de 16 niveles para almacenar los datos enviados por el procesador.

[4]Proporcionada por Xilinx con licencia libre

- Velocidad: incluye las velocidades en baudios más comunes.
- Bits de datos: desde 5 a 8 bits de datos por envío.
- Paridad: ninguna, par o impar.
- Bits de parada: 1, 1.5, 2. 1.5 no es soportado en sistemas UNIX.
- Sin control de flujo, por hardware y por software.

La configuración tendrá que ser de 8 bits para el campo de datos, sin paridad, con un bit de parada y sin control de flujo. Esto es debido a que el control de flujo no está soportado por la UART en las placas de evaluación Spartan 3E Starter Kit.

## 7.3. Fase de diseño

Habiendo establecido todos los requisitos necesarios para el sistema embebido, pasamos a la fase de diseño del sistema embebido.

En esta fase de diseño expondremos como hacer uso de los recursos de la placa de evaluación, el Picoblaze y el periférico receptor que también añadiremos al sistema. Además se expondrá la arquitectura del sistema indicando los buses y las conexiones entre los distintos elementos del mismo.

### 7.3.1. Picoblaze y arquitectura de memoria

Un sistema con el procesador software Picoblaze se basa en la arquitectura Harvard la cual posee una memoria para las instrucciones y otra para los datos. Para comunicar las dos memorias y el procesador se dispone de tres buses:

- Bus de dirección de instrucción: Tiene un ancho de 10 bits y selecciona la instrucción a leer de la memoria PROM de instrucciones.
- Bus de instrucción: Tiene un ancho de 18 bits. Son los 18 bits que componen una instrucción de Picoblaze.
- Bus de entrada datos: Tiene un ancho de 8 bits. Es el bus de la memoria de datos y de los periféricos de entrada/salida.
- Bus de salida datos: Tiene un ancho de 8 bits. Es el bus de la memoria de datos y de los periféricos de entrada/salida.

En el siguiente diagrama podemos ver tanto el diagrama de bloques del procesador como los buses que utiliza el procesador para acceder a la memoria de instrucciones y el bus para acceder a los datos.

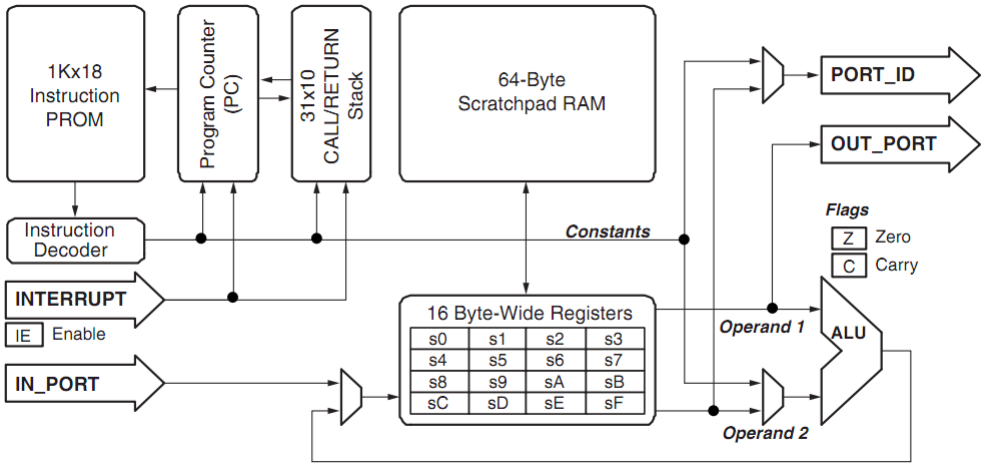


Figura 7.1: Diagrama de bloques del procesador Picoblaze

### 7.3.2. UART

Como se comentó en la fase de análisis, es necesaria una UART para enviar información recogida de las tramas al exterior. Para implementar una UART en la FPGA, Xilinx proporciona un módulo VHDL libre que implementa una UART muy básica y que para las necesidades del sistema embebido que se está desarrollando son más que suficientes.

El periférico irá conectado al bus de datos de salida del Picoblaze y la línea Tx será conectada al pin Tx del conector DB-9 de la placa de evaluación.

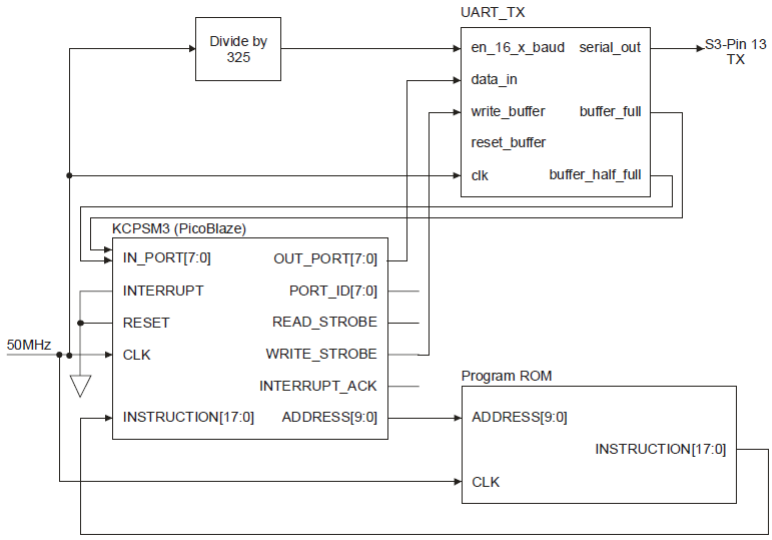


Figura 7.2: Diagrama de bloques de la UART

En este diagrama se puede observar la interconexión de todos los componentes que se han des-

crito individualmente en las fases de análisis y diseño. Se observa la arquitectura Harvard de memoria en la que la RAM embebida actúa de memoria de instrucciones, los registros de 32 bits del receptor de tramas son leídos por el Picoblaze de 8 en 8 bits mediante el multiplexor de 128 a 8. Esos 8 bits los lee el Picoblaze por el bus In_port.

Los datos se enviarán al exterior de 8 en 8 bits por la UART conectando ésta con el Picoblaze mediante el bus Out_port.

### 7.3.3. Interconexión de los componentes

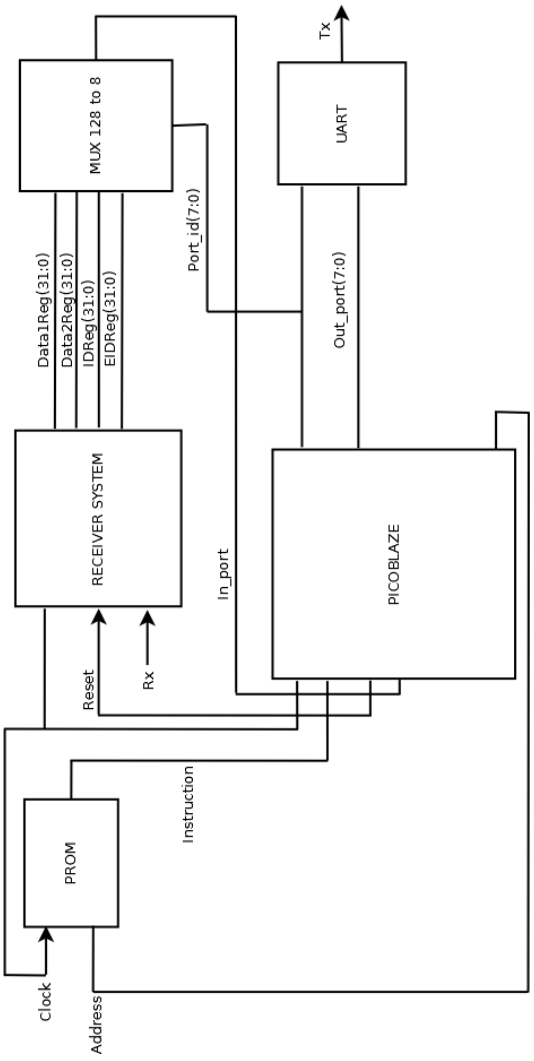


Figura 7.3: Diagrama del sistema embebido

## 7.4. Fase de implementación

En esta fase y en base a las especificaciones de diseño, se procedió a la implementación del sistema embebido usando para ello el *Project Navigator* de Xilinx.

### 7.4.1. Implementación con Xilinx *Project Navigator*

Para implementar el sistema embebido con Picoblaze, se debe crear un nuevo proyecto con *Project Navigator* estableciendo la placa de destino como "Spartan 3E starter kit" que será la que emplearemos durante este proyecto final de carrera.

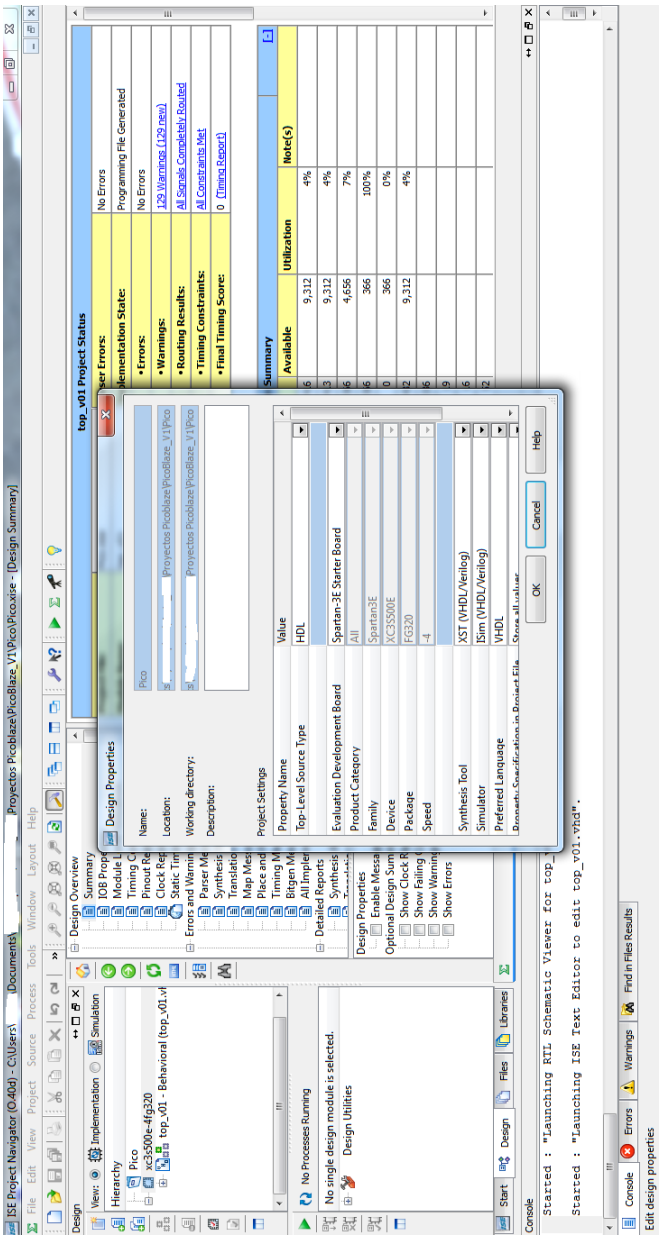

Figura 7.4: Implementación usando *Project Navigator*

Para generar el módulo superior del sistema embebido es necesario establecer la jerarquía de ficheros que van a componer cada elemento del sistema embebido. La jerarquía es la siguiente:

- Top.vhd: Fichero donde se instancian todos los componentes del sistema embebido.

- PROM.vhd: Fichero que contiene la descripción en VHDL de la memoria de programa.
- Uart_tx.vhd: Fichero que instancia los dos componentes de la uart que provee Xilinx. Estos son bbfifo_16x8.vhd que implementa la cola fifo de la UART y kcuart_tx.vhd que implementa la UART.
- Baud_gen.vhd: Generador de baudios para la UART.
- RECEIVER_SYSTEM.vhd: Receptor de tramas desarrollado en el capítulo anterior de este proyecto final de carrera.

En la siguiente figura podemos observar como queda la jerarquía al importar los ficheros al proyecto del *project navigator*

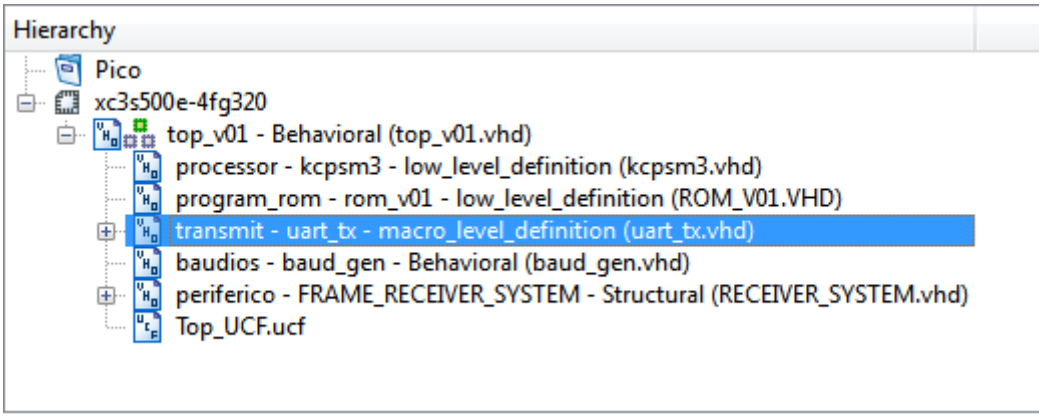


Figura 7.5: Jerarquía de ficheros del sistema embebido con procesador Picoblaze

### Módulo Picoblaze

La declaración de la entidad Picoblaze mostrando sus entradas y salidas es la siguiente:

```
entity kcpsm3 is
    Port (      address : out std_logic_vector(9 downto 0);
            instruction : in std_logic_vector(17 downto 0);
                port_id : out std_logic_vector(7 downto 0);
           write_strobe : out std_logic;
               out_port : out std_logic_vector(7 downto 0);
            read_strobe : out std_logic;
                in_port : in std_logic_vector(7 downto 0);
              interrupt : in std_logic;
          interrupt_ack : out std_logic;
                  reset : in std_logic;
                    clk : in std_logic);
end kcpsm3;
```

### Módulo UART

La declaración de la entidad UART y todos sus componentes internos es la siguiente:

```
--Entidad UART
entity uart_tx is
    Port (            data_in : in std_logic_vector(7 downto 0);
                 write_buffer : in std_logic;
                 reset_buffer : in std_logic;
                 en_16_x_baud : in std_logic;
                   serial_out : out std_logic;
                  buffer_full : out std_logic;
              buffer_half_full : out std_logic;
                           clk : in std_logic);
end uart_tx;
--Entidad Generador de baudios
entity baud_gen is
    Port ( clk : in  STD_LOGIC;
           en_16x_baud : out  STD_LOGIC);
end baud_gen;
--Entidad cola fifo
entity bbfifo_16x8 is
    Port (       data_in : in std_logic_vector(7 downto 0);
                data_out : out std_logic_vector(7 downto 0);
                   reset : in std_logic;
                   write : in std_logic;
                    read : in std_logic;
                    full : out std_logic;
               half_full : out std_logic;
            data_present : out std_logic;
                     clk : in std_logic);
    end bbfifo_16x8;
```

### Memoria PROM

La memoria PROM es la que contiene el programa que ejecutará el procesador Picoblaze.

Este programa escrito en ensamblador, ejecuta una rutina que lee una trama de los cuatro registros del periférico receptor de tramas, y posteriormente la envía al buffer de envío de la UART para que esta la envíe al exterior. Los resultados serán visualizados en un terminal con caracteres ascii.

## 7.5. Fase de pruebas

La fase de pruebas del sistema embebido tiene como objetivo comprobar que todos los componentes del sistema embebido funcionaban correctamente.

Para ello se implementó el diseño del sistema embebido sobre la placa de evaluación Spartan 3E Starter Kit. Puesto que para que el receptor de tramas funcione necesita recibir tramas CAN, se ha desarrollado un emisor de tramas CAN que será conectado al receptor.

Para probar que la rutina funciona correctamente, se pone en marcha tanto el sistema embebido como el emisor de tramas. Luego se conecta mediante el puerto serie, el sistema embebido con un PC. Este PC tendrá abierta una terminal para recibir la información que envíe el sistema embebido que como se comentó anteriormente serán caracteres ascii. Si el terminal soporta visualización en hexadecimal se debe comprobar que la información recibida coincide con la esperada.

# Capítulo 8

# Desarrollo del sistema embebido con procesador Microblaze de Xilinx

## 8.1. Introducción

En este capítulo se expondrá todo el proceso de desarrollo que se ha seguido para poner en funcionamiento un sistema embebido basado en el microprocesador software Microblaze. Este sistema embebido incluirá una serie de recursos entre los que estará el periférico receptor de tramas del capítulo anterior. El sistema embebido tendrá la funcionalidad de un sistema de adquisición de datos que podrá comunicarse con el exterior tanto usando elementos de la propia placa de evaluación como pudiéndose conectar a un ordenador personal para recibir los datos.

## 8.2. Fase de análisis

En esta fase vamos a analizar los requisitos que tenemos que cumplir a la hora de diseñar el sistema embebido tanto a nivel del periférico receptor como de los propios recursos que presenta la FPGA.

Analizaremos primero qué elementos necesitamos para diseñar un sistema embebido que tenga la funcionalidad que hemos comentado en la introducción. A continuación proporcionamos una lista de los elementos que va a necesitar nuestro sistema embebido y que posteriormente comentaremos en sus secciones correspondientes.

- Procesador software Microblaze de Xilinx [1].
- Memoria Block-Ram (Recurso de la FPGA).
- 4 interruptores (Recurso de la placa de evaluación).
- 8 leds (Recurso de la placa de evaluación).
- Pantalla LCD (Recurso de la placa de evaluación).

[1] IP Core propietario ofrecido por Xilinx, con el que se cuenta gracias a la licencia adquirida junto al software de desarrollo.

- UART[2].
- Periférico receptor de tramas CAN (Desarrollado en el anterior capítulo de este proyecto).

### 8.2.1. Procesador software Microblaze de Xilinx

El procesador software es la parte más importante del sistema embebido ya que sin él no podrían funcionar el resto de los elementos del sistema. Se trata de un procesador que posee las siguientes características:

- Ancho de palabra de 32 bits.
- Tipo RISC.
- Big Endian.
- Caché configurable.
- Pipeline de 3 o 5 fases.
- 32 registros de 32 bits.
- Frecuencia: 50-75 MHz en Spartan 3 y hasta 307 MHz en la familia Virtex
- Posibilidad de cargar un núcleo de Linux.
- Código cerrado.

La razón de usar el Microblaze es que es el más empleado en la industria de los sistemas embebidos para FPGAs de Xilinx. Además cuenta con una gran potencia de cálculo y destaca por su flexibilidad ya que dispone de varias configuraciones entre las que destacan las siguientes:

- Tamaño mínimo: reduce la frecuencia y velocidad para ocupar menor espacio en la FPGA. La reducción de tamaño viene del uso del pipeline de 3 fases en contraposición a las dos siguientes configuraciones que usan el de 5 fases.
- Máxima frecuencia: implementa el Microblaze para trabajar a la máxima frecuencia posible para aumentar el rendimiento. Esta configuración aumenta el tamaño que ocupa en la FPGA.
- Máximo rendimiento: idéntica a la anterior pero acelera las operaciones del procesador usando multiplicadores hardware. El procesador ocupará más incluso que en la anterior configuración.

[2]IP Core proporcionado por Xilinx

Como en el caso de nuestro sistema la velocidad no es crítica, seleccionamos la configuración de **"tamaño mínimo"** que redunda en un menor coste y consumo de energía.

En cuanto a la frecuencia del sistema, las FPGA de Xilinx disponen de un recurso denominado DCM que es capaz de elevar/dividir la frecuencia de reloj de la placa de evaluación. En la placa de evaluación que se está usando en este proyecto el DCM es capaz de elevar la frecuencia de 50 MHz del cristal oscilador a 75 MHz. Por tanto 75 MHz será la frecuencia de trabajo del procesador Microblaze.

### 8.2.2. Memoria Block-Ram

En todo sistema embebido es necesario disponer de una determinada cantidad de memoria para almacenar el programa que ejecuta el procesador. En el sistema embebido que se va a implementar en este proyecto, se necesita una pequeña cantidad de memoria RAM para almacenar el programa que recogerá datos del periférico y los mostrará y enviará al exterior usando los recursos de la placa de evaluación.

En las FPGA de Xilinx, esta memoria puede ser de dos tipos:

- Block-RAM: son pequeños bloques de memoria de RAM presentes en las FPGA de Xilinx en mayor o menor medida dependiendo del modelo. Son memorias configurables, normalmente de puerto dual en sistemas con Microblaze, y su tamaño puede oscilar entre unos pocos kilobytes y una gran cantidad de megabytes.

- Memoria distribuida: es un tipo de memoria que se forma uniendo Slices para usar sus LUTS como memoria RAM. Es un tipo de memoria más lenta que la anterior pero si falta memoria block RAM puede usarse para almacenar otros datos como la configuración de inicio de la FPGA, etc...

En el caso de nuestro sistema embebido disponemos de limitaciones en cuanto a la cantidad de block RAM debido a que la FPGA que se va a utilizar dispone sólo de 32 KBytes de RAM. Esto no supone un problema puesto que serán suficientes para almacenar el código de la aplicación. En caso de querer ampliar las posibilidades de la aplicación y que ya no pudiese ser almacenada en la block-RAM, sería necesario incluir un controlador DDR-2 para usar una memoria RAM externa.

### 8.2.3. Interruptores de la placa de evaluación

El sistema embebido está pensado para que el usuario pueda seleccionar las tramas que se visualizarán en la pantalla LCD. La placa de evaluación usada dispone de cuatro interruptores que se serán utilizados para este propósito.

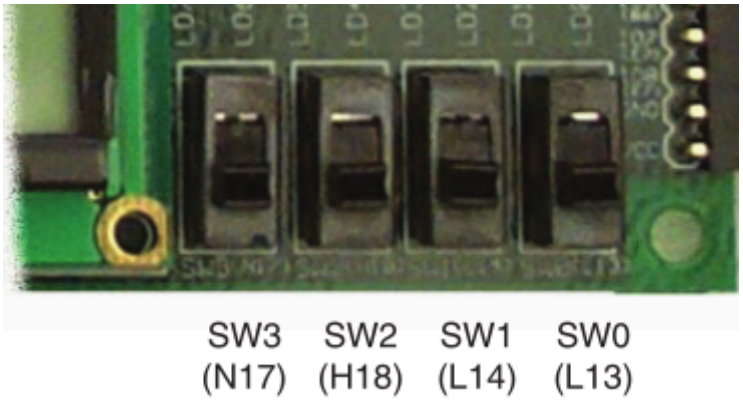


Figura 8.1: Interruptores de la placa de evaluación

### 8.2.4. Leds de la placa de evaluación

Durante el funcionamiento del sistema embebido, el usuario deberá conocer en todo momento si el sistema está funcionando adecuadamente, cuando está enviando y recibiendo peticiones. La placa de evaluación dispone para este propósito de ocho leds que serán controlados por el procesador.

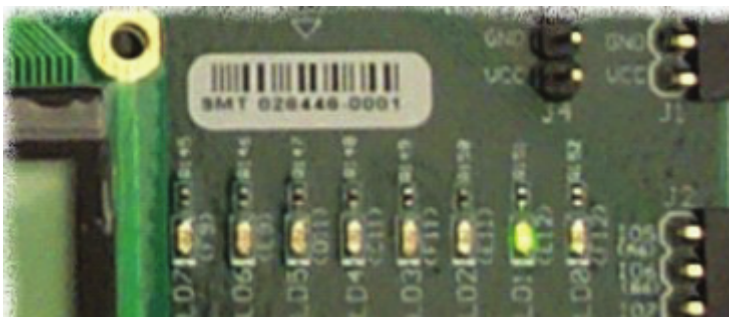

Figura 8.2: Leds de la placa de evaluación

### 8.2.5. Pantalla LCD de la placa de evaluación

La placa de evaluación dispone de una pantalla LCD de 16x2 que será utilizada para mostrar información al usuario. El procesador Microblaze escribirá en la pantalla la información relevante de las tramas recibidas. La trama mostrada dependerá de los interruptores que el usuario active. La placa de evaluación dispone de una pantalla LCD como la mostrada en la figura 8.3.


Figura 8.3: LCD de la placa de evaluación

### 8.2.6. UART

Además de usar la pantalla LCD de la placa de evaluación para comunicar el sistema embebido con el exterior, se implementará una UART en la FPGA conectada a uno de los conectores DB-9 que dispone la placa de evaluación para enviar datos a una aplicación de adquisición de datos[3]. La UART presenta los siguientes parámetros a configurar:

- Velocidad: incluye las velocidades en baudios más comunes.
- Bits de datos: desde 5 a 8 bits de datos por envío.
- Paridad: ninguna, par o impar.
- Bits de parada: 1, 1.5, 2. 1.5 no es soportado en sistemas UNIX.
- Sin control de flujo, por hardware y por software.

La configuración tendrá que ser de 8 bits para el campo de datos, sin paridad, con un bit de parada y sin control de flujo. Esto es debido al protocolo que posteriormente definiremos para comunicar la placa de evaluación con una aplicación software de adquisición de datos instalada ordenador personal. La placa de evaluación dispone de dos conectores DB-9 como los de la figura 8.4 para conectar las UARTs.

Figura 8.4: Conectores DB9 de la placa de evaluación

[3]Monitoring Tool. Ver capítulo 10

## 8.3. Fase de diseño

Habiendo establecido todos los requisitos necesarios para el sistema embebido, pasamos a la fase de diseño del sistema embebido.

En esta fase de diseño expondremos como hacer uso de los recursos de la placa de evaluación mediante los periféricos que nos proporciona el entorno de desarrollo EDK así como el periférico receptor que también añadiremos al sistema. Además se expondrá la arquitectura del sistema indicando los buses y las conexiones entre los distintos elementos del mismo.

### 8.3.1. Microblaze y arquitectura de memoria

Un sistema con el procesador software Microblaze a diferencia de la mayoría de los computadores de sobremesa, se basa en la arquitectura Harvard la cual posee una memoria para las instrucciones y otra para los datos. Realmente, en un sistema con procesador Microblaze la memoria de datos y de instrucciones es físicamente la misma debido a que esta memoria es de doble puerto y el Microblaze accede a las instrucciones mediante un bus y a los datos mediante otro. Ambos buses son de 32 bits por lo que se podrá direccionar hasta 4 GBytes de datos e instrucciones.

Hay que tener en cuenta que este tipo de memoria permite accesos simultáneos por ambos puertos a la vez mientras no se acceda a la misma zona de memoria. A la hora de cargar el programa en memoria, el cargador se encarga de separar las instrucciones de los datos.

En el siguiente diagrama podemos ver tanto el diagrama de bloques del procesador como los buses que utiliza el procesador para acceder a la memoria, el ILMB (Instruction Local Memory Bus) para acceder a las instrucciones y el DLMB (Data Local Memory Bus) para acceder a los datos.

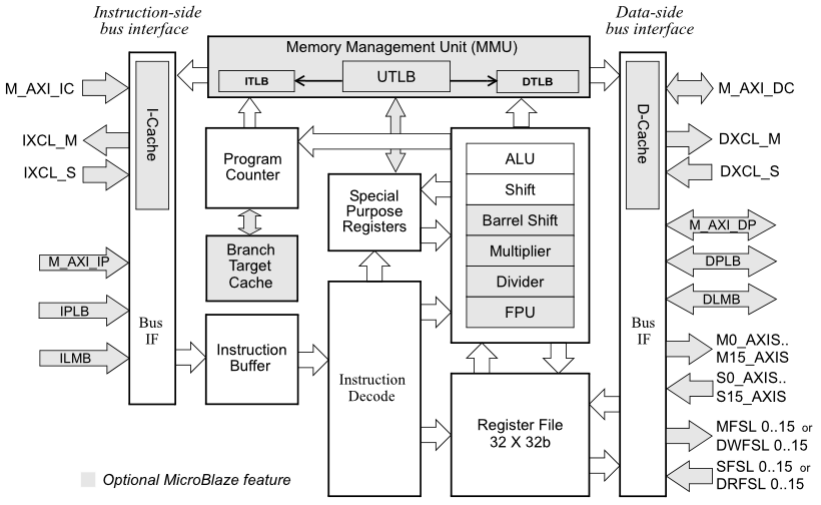


Figura 8.5: Diagrama de bloques del procesador Microblaze

### 8.3.2. Bus PLB

Anteriormente hemos visto como Microblaze gestiona la memoria. Falta por conocer como se comunica con sus periféricos. Para ello usa el bus PLB (Processor Local Bus) [4] que será el que compartan todos los periféricos. Los periféricos pueden ser conectados al bus tanto como maestros como esclavos como se puede observar en la figura 7.6

En el sistema embebido de este proyecto todos los periféricos serán conectados como esclavos puesto que será el Microblaze el que esté conectado como maestro para tener el control absoluto del bus. El conectarlos como esclavos no afectará en absoluto al rendimiento del sistema puesto que los periféricos que conectaremos al bus no requieren una velocidad de escritura/lectura demasiado alta.

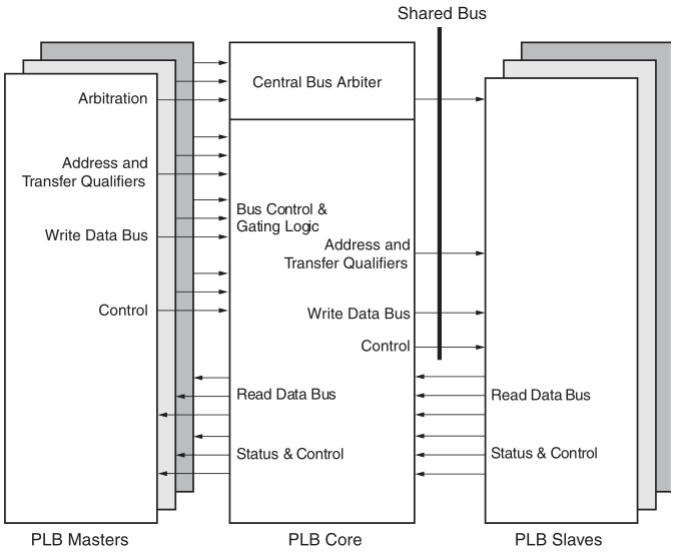


Figura 8.6: Bus PLB (Autor: Xilinx)

### 8.3.3. Interruptores

Los interruptores de la placa de evaluación serán los únicos dispositivos de entrada del sistema embebido. Para poder controlar los interruptores desde el código que ejecutará el procesador necesitamos un periférico que sea capaz de gestionar entradas y salidas. EDK [5] nos proporciona uno que se denomina GPIO (General Purpose Input Output).

GPIO es un periférico en el que disponemos de hasta dos canales configurables de 32 bits de ancho de palabra como máximo para conectar a pines de entrada/salida de la FPGA. El periférico dispone de una serie de registros en los que se almacenan los valores de las entradas y salidas. En estos registros se puede tanto escribir para asignarle un valor determinado a un pin de salida de la FPGA como leer para recoger el valor de una determinado pin de entrada.

[4] IP propietaria de Xilinx que es proveída junto a la licencia del software de desarrollo.
[5] Embedded Development Kit: IDE de Xilinx para sistemas embebidos

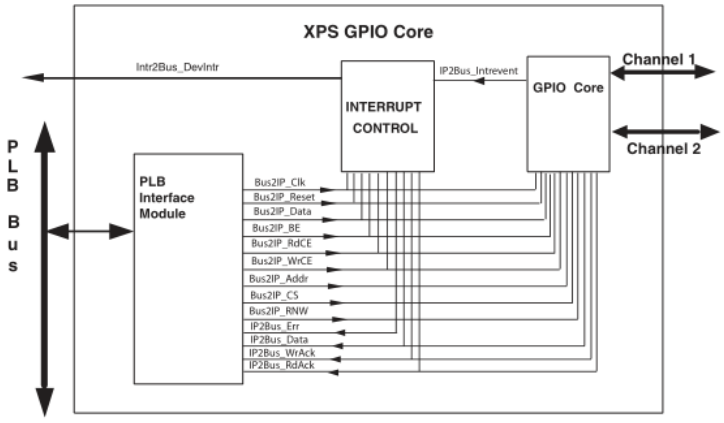


Figura 8.7: Diagrama de bloques de GPIO (Autor: Xilinx)

En lo que respecta al control de los interruptores, se hace uso del canal 1 del GPIO estableciendo su anchura en 4 bits puesto que sólo hay cuatro interruptores. Además como sólo se leerá de los registros, se establece el canal 1 como de sólo entrada. El canal 2 estará desactivado puesto que no se usará.

### 8.3.4. Leds

Al igual que en caso de los interruptores necesitamos un GPIO para el control de los leds de la placa de evaluación. A diferencia del anterior, los leds son salidas del sistema por lo que para su control debemos escribir en el GPIO. Como detallamos en la fase de análisis, la placa dispone de 8 leds por lo que debemos configurar el canal 1 con una anchura de 8 bits, uno por cada led. El canal estará desactivado debido a que no se usará.

### 8.3.5. Pantalla LCD y timer

#### Control del LCD con GPIO

La pantalla LCD es un dispositivo de salida y como tal debe ser controlado usando otro GPIO. La pantalla LCD de la placa de evaluación dispone de una interfaz de siete pines, cuatro de los cuales comparte con la memoria Intel StrataFlash.

En el siguiente diagrama podemos ver los pines de la FPGA y las señales para el control de la pantalla LCD.

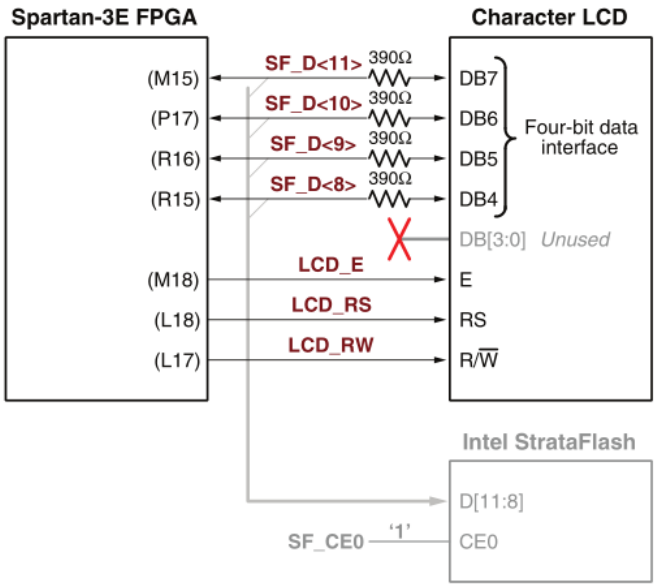


Figura 8.8: Diagrama de bloques de la pantalla LCD

En la siguiente tabla se recogen las señales de control del LCD y la función de cada una.

| Señal | Descripción |
|---|---|
| SF_D<11> | Bit de datos 7. |
| SF_D<10> | Bit de datos 6. |
| SF_D<9> | Bit de datos 5. |
| SF_D<8> | Bit de datos 4. |
| LCD_E | Lectura/Escritura. 0: Desactivado, 1: Operación E/S habilitada. |
| LCD_RS | Selector de registro. 0: Registro de instrucción, 1: Registro de datos. |
| LCD_RW | Control de lectura/escritura. 0: Escritura, 1: Lectura. |

Tabla 8.1: Pines de control de la pantalla LCD

Para el control del LCD se dispondrá un bloque GPIO. El canal 1 de mismo se configurará a 7 bits de anchura en el canal 2 será desactivado puesto que no se es necesario su uso. Los siete bits de los que se compone el canal 1 del GPIO serán las señales del LCD reflejadas en la anterior tabla. Mediante escrituras y lecturas con los valores deseados para las señales en el registro del GPIO enviaremos los datos y las instrucciones al LCD.

Tanto las instrucciones para controlar la pantalla como los datos tienen una longitud de ocho bits. La pantalla LCD dispone de cuatro líneas de datos por lo que son necesarias dos escrituras de cuatro bits. La primera escribirá los cuatro bits más significativos y la segunda el resto de los bits. Entre las dos escrituras es necesario esperar una pequeña cantidad de tiempo.

**Timer de control**

Tanto para las esperas entre escrituras como para otras instrucciones de control de la pantalla LCD será necesario usar un timer que el entorno de desarrollo EDK nos provee con el bloque "xps_timer". El timer tendrá que configurarse para permitir una resolución de microsegundos debido a que será el tiempo mínimo que debemos esperar entre dos escrituras consecutivas en los registros del LCD.

Otra función del timer es el control del proceso de inicialización de la pantalla LCD. En la sección del desarrollo del programa software se expondrá todo el proceso de inicialización y control del LCD indicando los retardos necesarios.

### 8.3.6. UART

Como se comentó en la fase de análisis, es necesaria una UART para enviar información recogida de las tramas al exterior. Para implementar una UART en la FPGA, el entorno de desarrollo EDK proporciona un bloque denominado "xps_uartlite" que implementa una UART muy básica y que para las necesidades del sistema embebido que se está desarrollando son más que suficientes. La figura 7.9 representa la estructura del bloque xps_uartlite que implementa la UART que se usará para las comunicaciones del sistema embebido con el exterior.

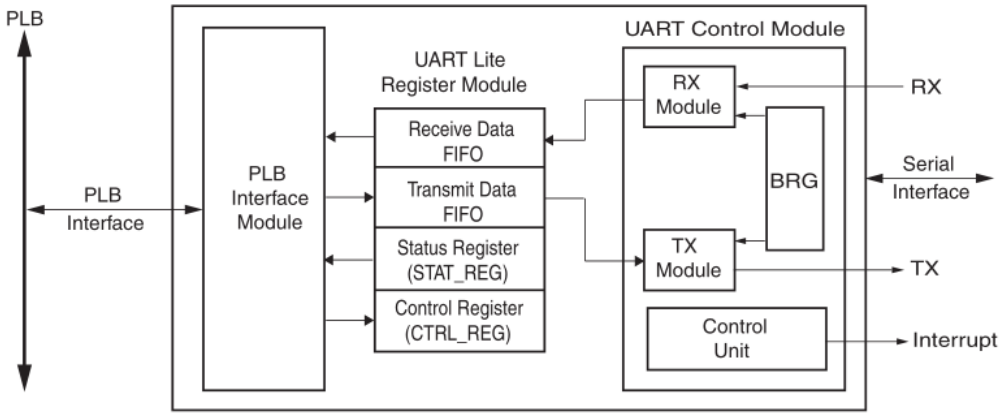


Figura 8.9: Diagrama de bloques de xps_uartlite (Autor: Xilinx)

Como podemos observar en el diagrama, el periférico irá conectado al bus PLB como esclavo al igual que el resto de los periféricos y las líneas Rx y Tx serán conectadas a los pines Rx y Tx del conector DB-9 de la placa de evaluación.

### 8.3.7. Interconexión de los componentes

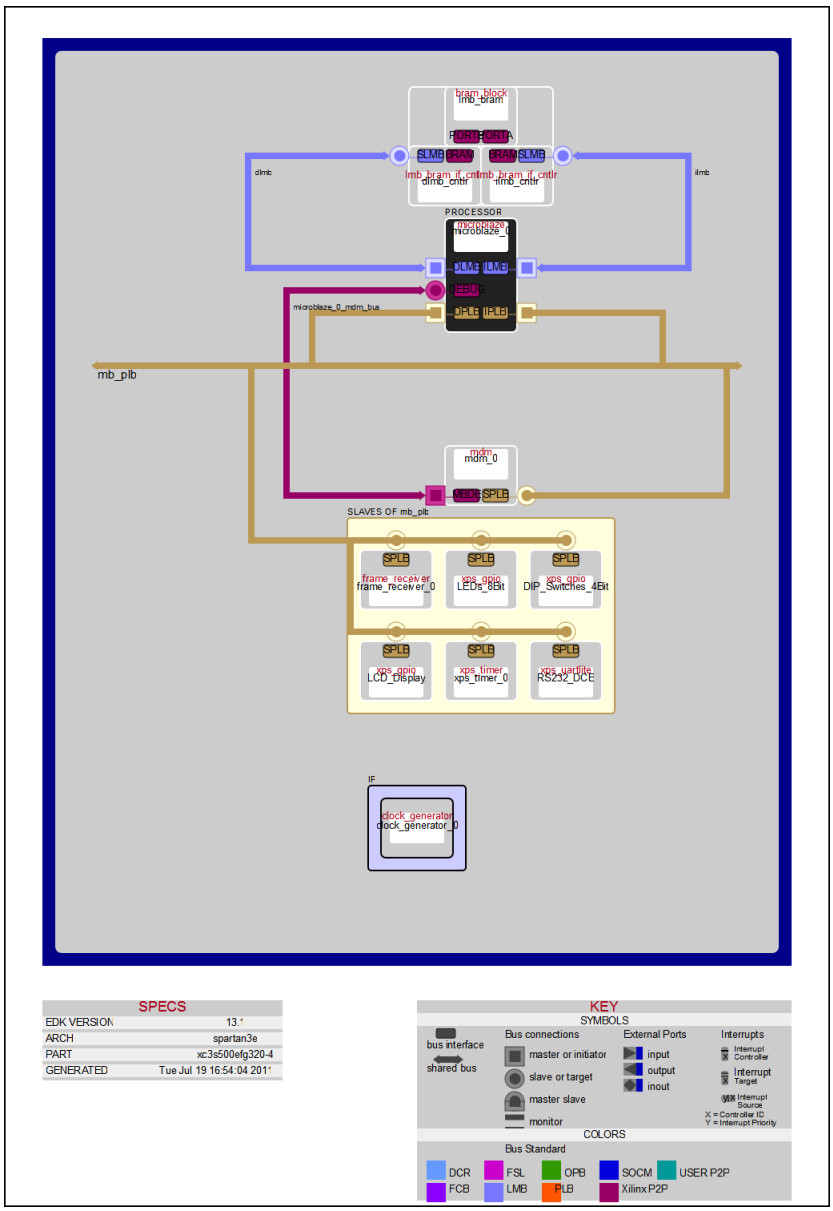


Figura 8.10: Diagrama del sistema embebido

En este diagrama se puede observar la interconexión de todos los componentes que se han descrito individualmente en las fases de análisis y diseño. Se observa la arquitectura Harvard así como la presencia de un módulo de depuración conectado al procesador que servirá para introducir puntos de ruptura en el software y poder depurarlo dentro del entorno de desarrollo de software SDK para Microblaze.

### 8.3.8. Mapa de memoria

Un sistema embebido con Microblaze mapea todos sus periféricos con direcciones de memoria del sistema. Es necesario establecer un rango de direcciones para cada periférico para luego en el software hacer referencia a ellos.
En la siguiente tabla queda recogido el mapa de memoria del sistema:

| Periférico | Dirección base | Desplazamiento | Tamaño | Bus |
|---|---|---|---|---|
| dlmb_cntlr | 0x00000000 | 0x7FFF | 32 K | LMB |
| ilmb_cntlr | 0x00000000 | 0x7FFF | 32 K | LMB |
| Leds_8Bits | 0x81400000 | 0xFFFF | 64 K | PLB |
| LCD_Display | 0x81420000 | 0xFFFF | 64 K | PLB |
| Switches_4Bits | 0x81440000 | 0xFFFF | 64 K | PLB |
| Xps_timer | 0x83C00000 | 0xFFFF | 64 K | PLB |
| RS232_DCE | 0x84000000 | 0xFFFF | 64 K | PLB |
| Mdm_0 | 0x84400000 | 0xFFFF | 64 K | PLB |

Tabla 8.2: Mapa de memoria del sistema embebido

A continuación se describen los anteriores componentes:

- dlmb_cntlr: Controlador de datos de la memoria RAM del sistema embebido. Está conectado al **Local Memory Bus** que es el bus de la memoria RAM conectada al Microblaze.
- dlmb_cntlr: Controlador de instrucciones de la memoria RAM del sistema embebido. Está conectado al **Local Memory Bus**.
- Leds_8Bits: Controlador de los leds de la placa de evaluación implementado con un GPIO. Está conectado al **Processor Local Bus** que es el bus que comparten todos los periféricos conectados al Microblaze.
- LCD_Display: Controlador de la pantalla LCD de la placa de evaluación implementado con un GPIO. Está conectado al **Processor Local Bus**.
- Switches_4Bits: Controlador de los interruptores de la placa de evaluación implementado con un GPIO. Está conectado al **Processor Local Bus**.
- Xps_timer: Timer que controla los retardos de la pantalla LCD de la placa de evaluación. Está conectado al **Processor Local Bus**.
- RS232_DCE: Controlador de la UART de la placa de evaluación. Está conectado al **Processor Local Bus**.
- Mdm_0: Depurador del Microblaze que permite insertar puntos de ruptura en los programas desde el software de desarrollo SDK. Está conectado al **Processor Local Bus**.

## 8.4. Fase de implementación

En esta fase y en base a las especificaciones de diseño, se procedió a la implementación del sistema embebido usando para ello el entorno de desarrollo para sistemas embebidos de Xilinx. Xilinx Platform Studio provee todos los elementos que fueron comentados en la fase de diseño además de asistir la implementación de un sistema embebido basado en Microblaze con su interfaz completamente visual, lo cual reduce en gran cantidad el tiempo de implementación. Es por ello que Xilinx Platform es el entorno de desarrollo más utilizado en la industria electrónica para los sistemas embebidos en FPGAs Xilinx.

Además del uso del entorno de desarrollo, será necesario describir en VHDL el manejador del periférico receptor de tramas para que pueda ser conectado al bus PLB, ya que se diseñó con la filosofía de ser compatible con cualquier tipo de bus y que fuera el usuario final el que lo adaptase al bus que fuera a utilizar.

### 8.4.1. Implementación con Xilinx Platform Studio

Para implementar el sistema embebido usaremos el entorno de desarrollo de Xilinx destinado para ello.

Al entrar en la aplicación obtendremos una ventana emergente en la que seleccionaremos **Base system builder**

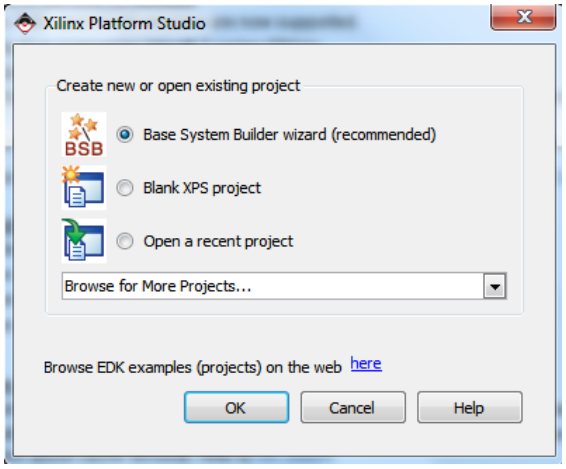


Figura 8.11: Base System Builder (Autor: Xilinx)

Una vez aceptemos nos pedirá un directorio donde almacenar el sistema embebido. El siguiente paso es establecer el tipo de interconexión entre los componentes. Como se ha comentado en las anteriores fases, el sistema se basará en el bus PLB©por lo que lo seleccionaremos.

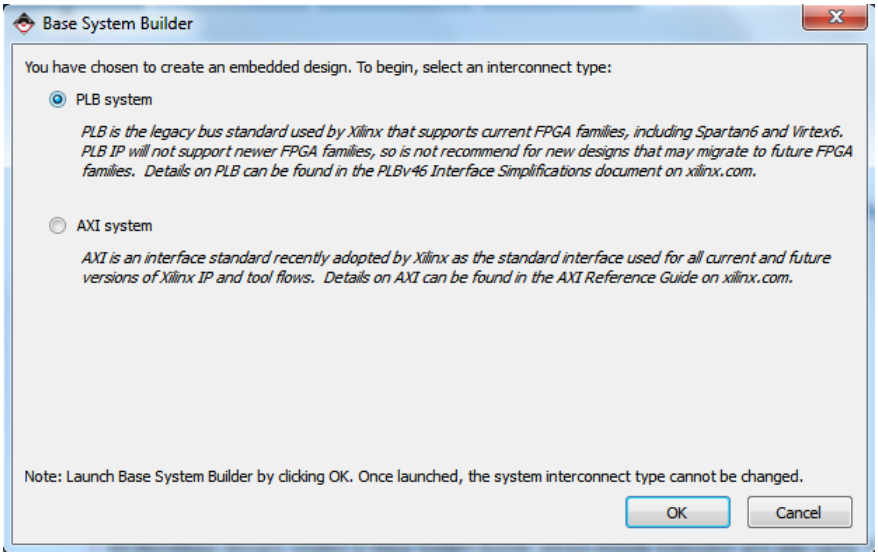


Figura 8.12: Plb system (Autor: Xilinx)

Seleccionamos la placa de evaluación en la que se va a implementar el sistema embebido. En el caso de este proyecto la placa de evaluación elegida es la Spartan 3E Starter Kit.

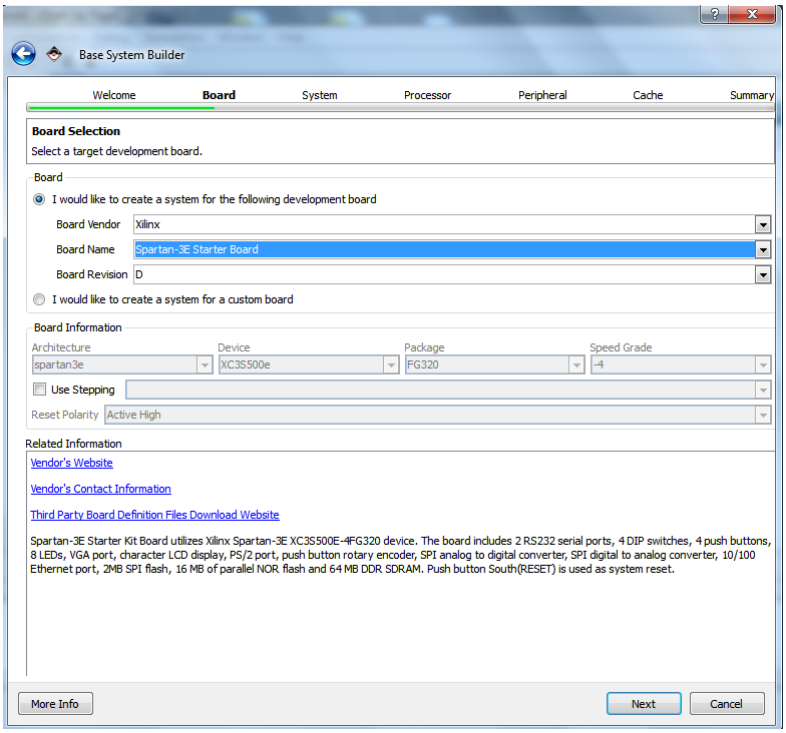


Figura 8.13: Selección de placa de evaluación (Autor: Xilinx)

En las siguientes ventanas configuraremos las opciones del sistemas embebido. La primera de ellas nos pedirá el número de procesadores del sistema. En la fase de análisis establecimos que sólo seria necesario un procesador Microblaze.

En la siguiente pantalla de configuración establecemos la frecuencia de trabajo del procesador y la cantidad de memoria RAM que tendrá el sistema embebido. En la fase de análisis se estableció que la frecuencia sería de 75 MHz y la memoria RAM de 32 KBytes de capacidad

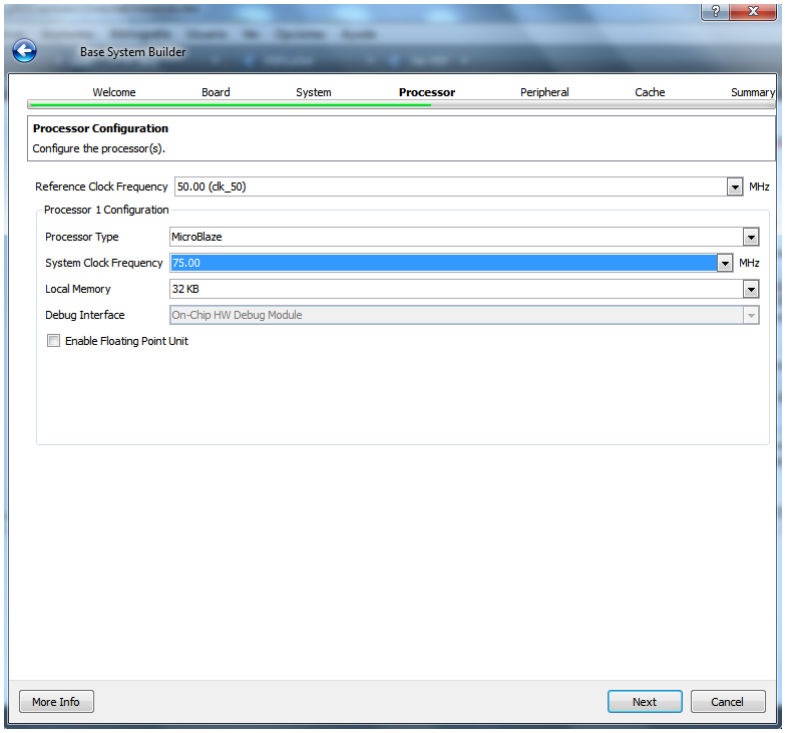


Figura 8.14: Selección de frecuencia y RAM (Autor: Xilinx)

La siguiente pantalla nos permitirá añadir o eliminar periféricos básicos al sistema embebido. En esta pantalla se añaden tanto las GPIO para los leds e interruptores, la UART con la configuración que deseemos (en este caso con una velocidad de 9600 baudios) y los controladores de memoria RAM.

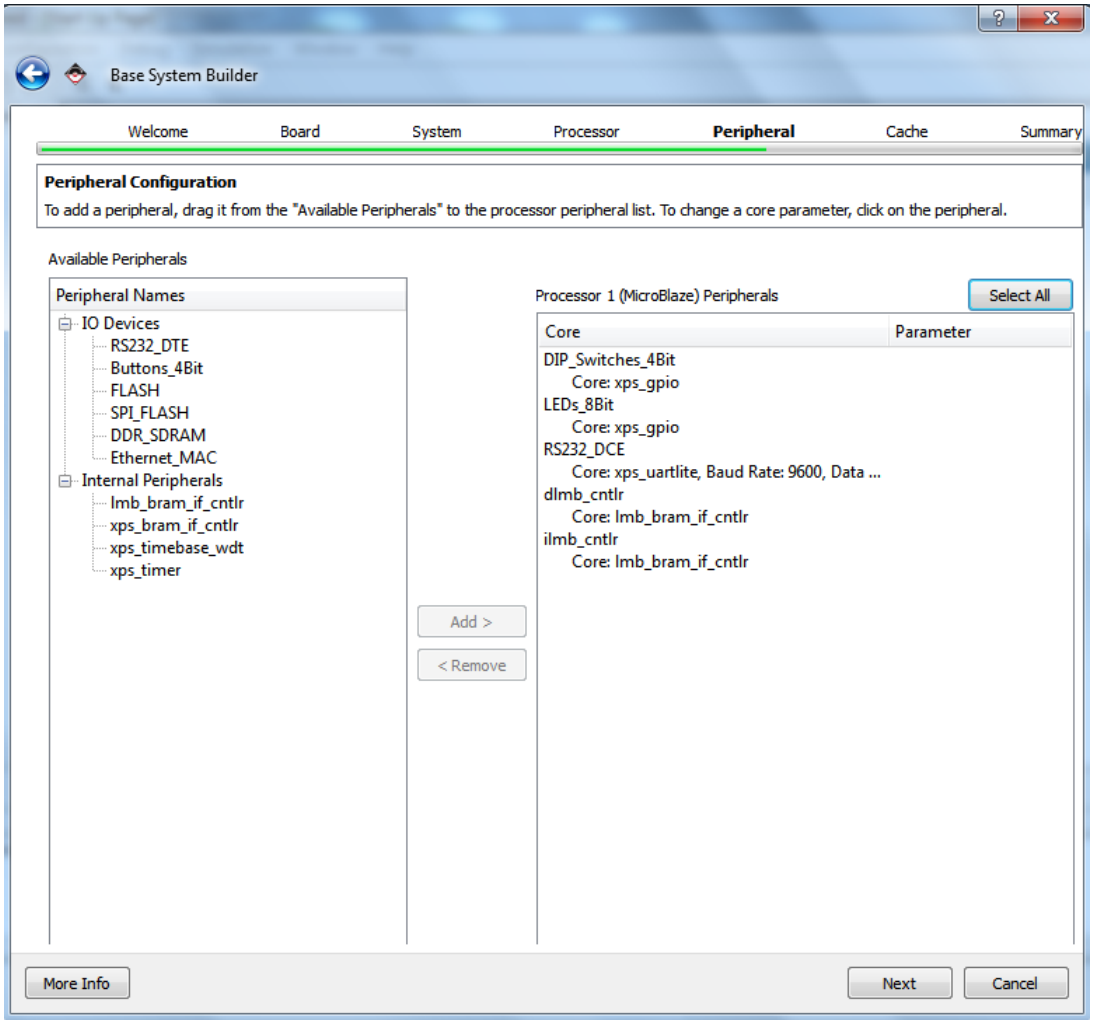


Figura 8.15: Periféricos del sistema embebido (Autor: Xilinx)

Para añadir los periféricos restantes al sistema los seleccionamos del catálogo IP y los arrastramos al sistema. Los periféricos restantes son el Xps_timer para el LCD, el frame_receiver que es el periférico receptor de tramas implementado en este proyecto con su manejador para el bus PLB y un GPIO para el control del LCD.

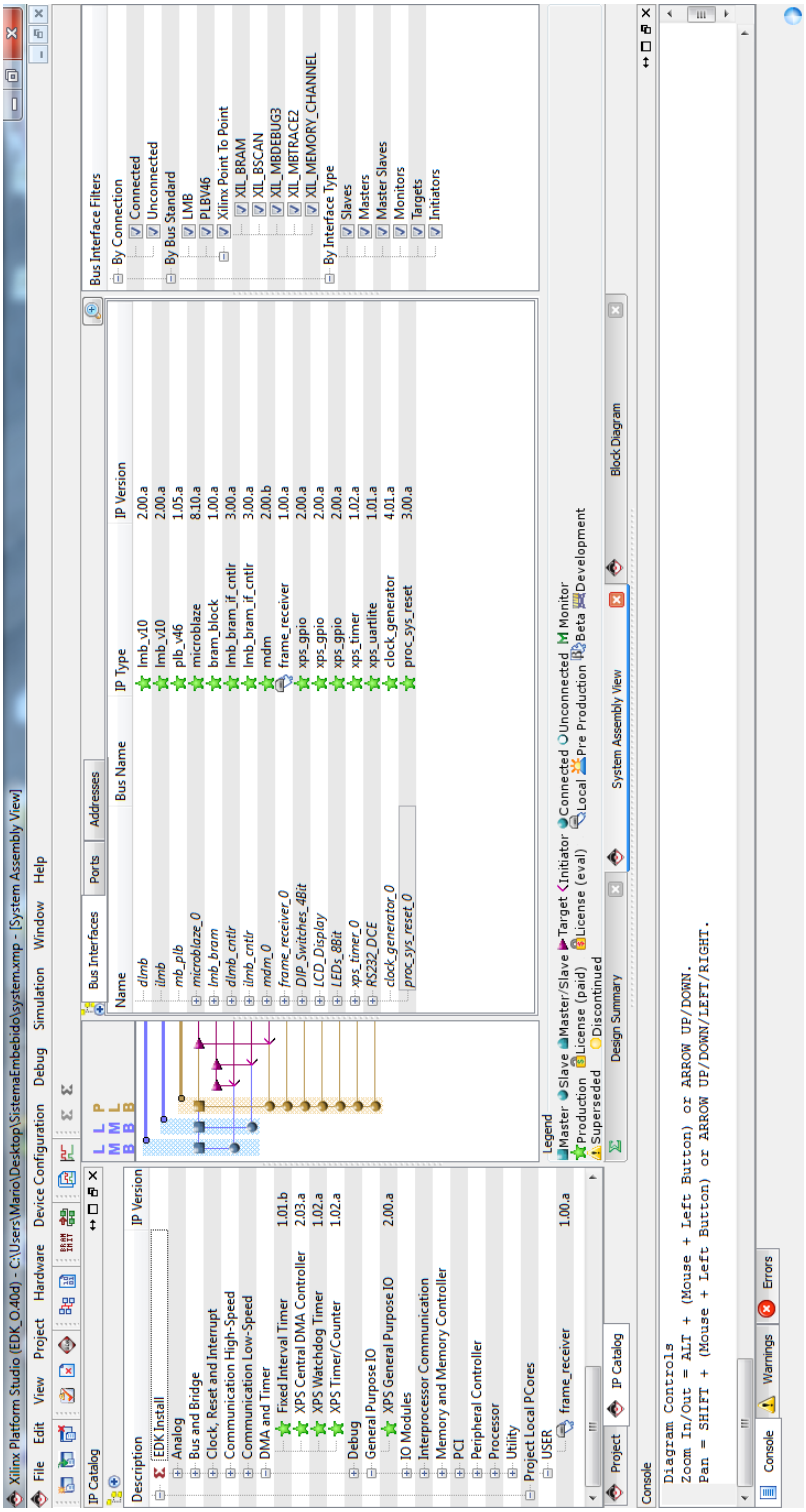

Figura 8.16: Añadir periféricos al sistema embebido (Autor: Xilinx)

La asignación de puertos a los pines de la FPGA se realiza desde el siguiente menú estableciendo los pines de la FPGA a los que se conectarán las entradas del periférico receptor de tramas y del GPIO de control del LCD.

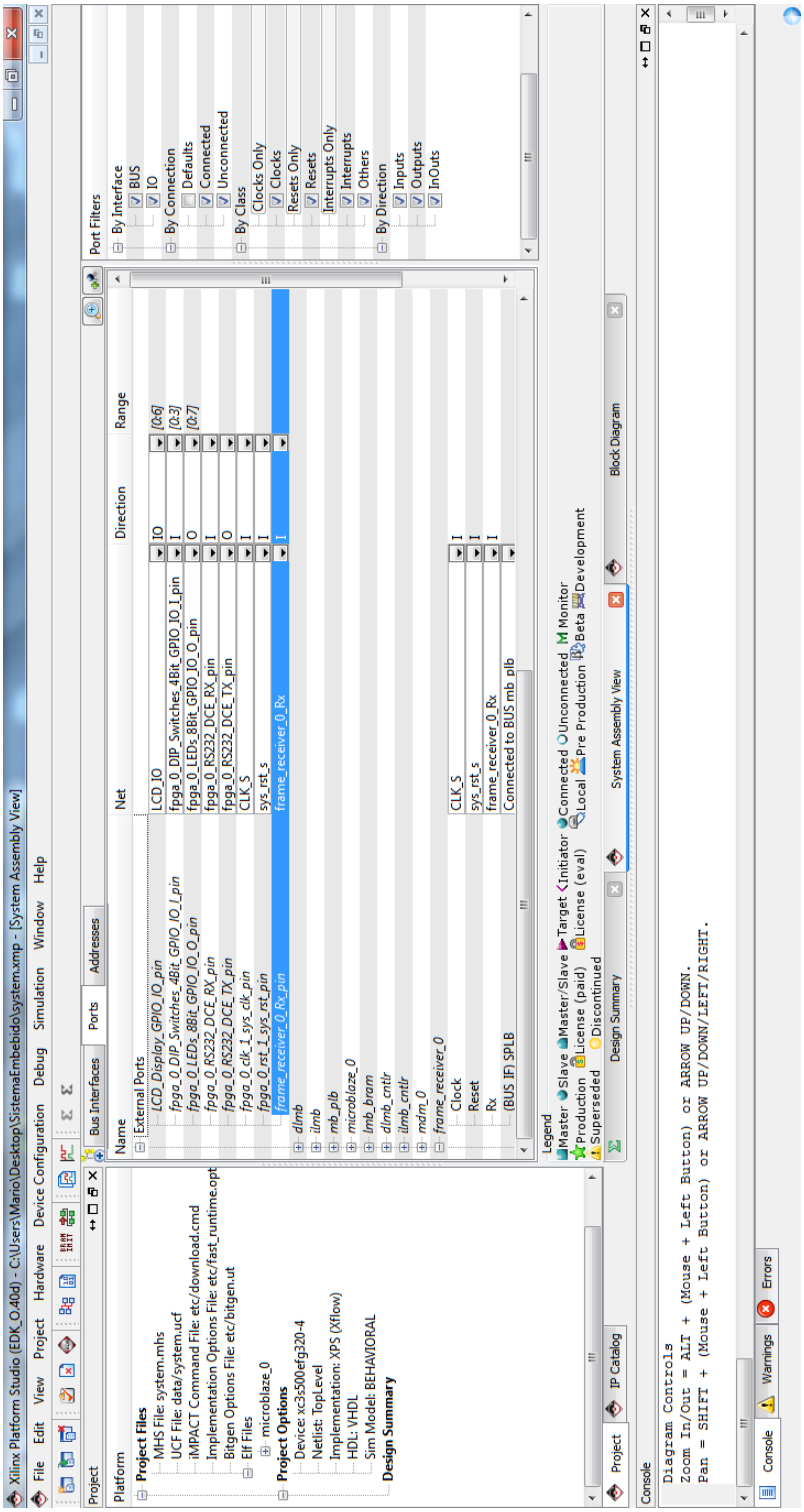

Figura 8.17: Añadir periféricos al sistema embebido (Autor: Xilinx)

El mapeo de la memoria se realiza desde el siguiente menú. En el estableceremos el mapa de memoria que se estableció en la fase de diseño del sistema embebido.

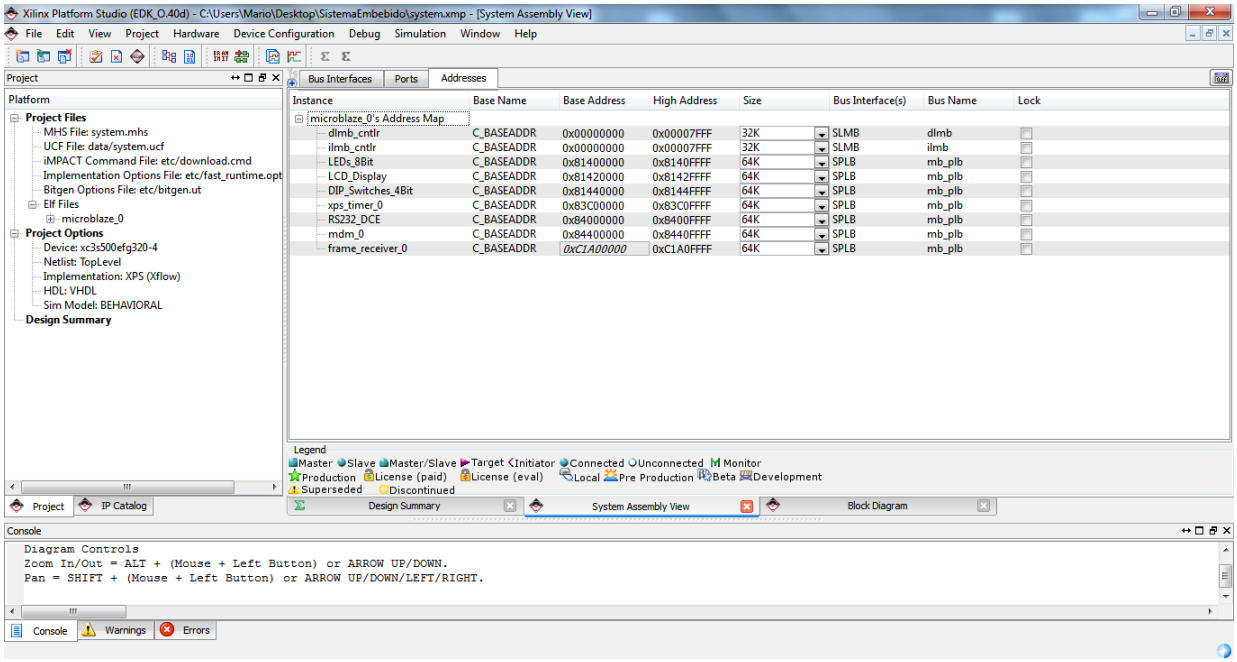


Figura 8.18: Mapa de memoria del sistema embebido (Autor: Xilinx)

Una vez esté todo el sistema embebido configurado de acuerdo a las especificaciones de análisis y diseño se debe generar el sistema embebido y exportar su diseño a la herramienta de desarrollo de software SDK. Debemos seguir los siguientes pasos:

1. Generar la netlist: Fase de síntesis el sistema embebido.

2. Generar el Bitstream: Fase de emplazamiento y ruteado del sistema embebido.

3. Exportar el diseño al SDK [6].

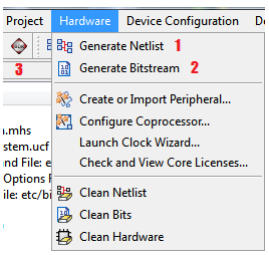


Figura 8.19: Fases de implementación del sistema embebido (Autor: Xilinx)

[6]Software Delopment Kit: IDE para programar sistemas embebidos con C/C++ de Xilinx

### 8.4.2. Implementación del manejador del periférico receptor para el bus PLB

La implementación del manejador para el bus PLB se compone de dos tareas principales:

1. Generación de la plantilla del manejador usando la utilidad del entorno de desarrollo XPS para crear un nuevo periférico para el bus PLB.

2. Modificación de la plantilla generada para conectar el manejador con la interfaz que posee el receptor de tramas.

#### Generación de la plantilla

Para generar la plantilla debemos ejecutar el asistente para la creación de un nuevo periférico. Una vez arrancado marcamos la opción creación de un nuevo periférico y le ponemos el nombre "receiver". Cuando se solicite la interfaz para la cual se quiere crear el nuevo periférico seleccionaremos **Processor Local Bus**. Puesto que el periférico no usará una cola FIFO, sólo se marcará la opción **sotfware user registers** para crear un periférico con registros. El número de registros que tendrá la plantilla será cuatro, tal y como se especificó en la fase de diseño del receptor de tramas.

#### Modificación de la plantilla

Una vez tenemos generada la plantilla, aparecerán tres ficheros en el directorio de nuestro periférico que debemos editar:

- pcores/receiver_1_00_a/hdl/vhdl/receiver.vhd: Este fichero es el manejador que conectará el periférico al bus PLB. Aquí debemos mapear los puertos que tenga el diseño del periférico receptor de tramas.

- pcores/receiver_1_00_a/hdl/vhdl/receiver.vhd: Este fichero es donde debe ir la lógica de funcionamiento del periférico.

- pcores/receiver_1_00_a/data/receiver_v2_1_0.pao: Este fichero es el que establece la jerarquía en el diseño VHDL del periférico. Debe ser editado para indicar la jerarquía que forman los dos ficheros anteriores junto con los del periférico.

#### Fichero receiver.vhd

Comenzamos por el fichero **"receiver.vhd"** en el que tenemos que añadir unas líneas con los puertos del periférico receptor en dos puntos distintos del fichero.

Para añadir los puertos editamos el fichero como sigue. Estos serán los puertos del periférico que se conectarán a los pines de la FPGA para obtener la señal de reloj, el reset y el Rx.

```
entity receiver is
  generic
  (
    -- ADD USER GENERICS BELOW THIS LINE ---------------
    --USER generics added here
    -- ADD USER GENERICS ABOVE THIS LINE ---------------

    -- DO NOT EDIT BELOW THIS LINE ---------------------
    -- Bus protocol parameters, do not add to or delete
    C_BASEADDR                     : std_logic_vector     := X"FFFFFFFF";
    C_HIGHADDR                     : std_logic_vector     := X"00000000";
    C_SPLB_AWIDTH                  : integer              := 32;
    C_SPLB_DWIDTH                  : integer              := 128;
    C_SPLB_NUM_MASTERS             : integer              := 8;
    C_SPLB_MID_WIDTH               : integer              := 3;
    C_SPLB_NATIVE_DWIDTH           : integer              := 32;
    C_SPLB_P2P                     : integer              := 0;
    C_SPLB_SUPPORT_BURSTS          : integer              := 0;
    C_SPLB_SMALLEST_MASTER         : integer              := 32;
    C_SPLB_CLK_PERIOD_PS           : integer              := 10000;
    C_INCLUDE_DPHASE_TIMER         : integer              := 1;
    C_FAMILY                       : string               := "virtex6"
    -- DO NOT EDIT ABOVE THIS LINE ---------------------
  );
port
  (
    -- ADD USER PORTS BELOW THIS LINE ------------------
      CAN_CLK => CAN_CLK,
      Rst => Rst,
      Rx => Rx,
    -- ADD USER PORTS ABOVE THIS LINE ------------------
```

Para mapear los puertos editamos el fichero con las siguientes líneas.

```
------------------------------------------
  -- instantiate User Logic
  ------------------------------------------
  USER_LOGIC_I : entity can_v1_00_a.user_logic
    generic map
    (
      -- MAP USER GENERICS BELOW THIS LINE ---------------
      --USER generics mapped here
      -- MAP USER GENERICS ABOVE THIS LINE ---------------

      C_SLV_DWIDTH                   => USER_SLV_DWIDTH,
      C_NUM_REG                      => USER_NUM_REG
    )
    port map
```

```
    (
      -- MAP USER PORTS BELOW THIS LINE ------------------
      CAN_CLK => CAN_CLK,
      Rst => Rst,
      Rx => Rx,
      --USER ports mapped here
      -- MAP USER PORTS ABOVE THIS LINE ------------------
```

**Fichero user_logic.vhd**

En este fichero añadiremos tantos los puertos que mapeamos en el fichero anterior como la lógica del periférico.

Añadimos los puertos en la siguiente línea:

```
entity user_logic is
  generic
  (
    -- ADD USER GENERICS BELOW THIS LINE ---------------
    --USER generics added here
    -- ADD USER GENERICS ABOVE THIS LINE ---------------

    -- DO NOT EDIT BELOW THIS LINE ---------------------
    -- Bus protocol parameters, do not add to or delete
    C_SLV_DWIDTH                   : integer              := 32;
    C_NUM_REG                      : integer              := 4
    -- DO NOT EDIT ABOVE THIS LINE ---------------------
  );
  port
  (
    -- ADD USER PORTS BELOW THIS LINE ------------------
    CAN_CLK                    : in std_logic;
    Rst                        : in std_logic;
    Rx                         : in std_logic;
    --USER ports added here
    -- ADD USER PORTS ABOVE THIS LINE ------------------
```

Para añadir la lógica se instancia el componente receptor:

```
------------------------------------------------------------------------------
-- Architecture section
------------------------------------------------------------------------------

architecture IMP of user_logic is

  --USER signal declarations added here, as needed for user logic
```

```
  component CAN_SYSTEM
   port ( Clk            : in    std_logic;
          Reset          : in    std_logic;
          Rx             : in    std_logic;
          Data1          : out STD_LOGIC_VECTOR (31 downto 0);
          Data2          : out STD_LOGIC_VECTOR (31 downto 0);
          Id             : out STD_LOGIC_VECTOR (31 downto 0);
          EId            : out STD_LOGIC_VECTOR (31 downto 0));
  end component;
```

Lo siguiente que debemos conectar son los registros de la plantilla del periférico a los de la interfaz del receptor. Para hacerlo debemos añadir cuatro vectores de 32 bits de tipo std_logic_vector que mapearemos en el componente definido en el anterior paso.

```
--CAN REGISTER
  signal IDREG:std_logic_vector(0 to 31);
  signal D1REG:std_logic_vector(0 to 31);
  signal D2REG:std_logic_vector(0 to 31);
  signal IDEREG:std_logic_vector(0 to 31);

begin

  --USER logic implementation added here
  CAN: CAN_SYSTEM
    port map( CAN_CLK,
          Rst,
          Rx,
                  D1REG,
                  D2REG,
                  IDREG,
                  IDEREG);

  -- implement slave model software accessible register(s) read mux
  SLAVE_REG_READ_PROC : process( slv_reg_read_sel,D1REG,D2REG,IDREG,IDEREG ) is
  begin

    case slv_reg_read_sel is
      when "1000" => slv_ip2bus_data <= IDREG;
      when "0100" => slv_ip2bus_data <= D2REG;
      when "0010" => slv_ip2bus_data <= D1REG;
      when "0001" => slv_ip2bus_data <= IDEREG;
      when others => slv_ip2bus_data <= (others => '0');
    end case;

  end process SLAVE_REG_READ_PROC;
```

**Fichero receiver_v2_1_0.pao**

Para establecer la jerarquía de ficheros VHDL del periférico debemos editar este fichero de la siguiente forma:

```
##############################################################################
## Filename:          C:\CAN\pcores/can_v1_00_a/data/can_v2_1_0.pao
## Description:       Peripheral Analysis Order
## Date:              Wed May 11 13:48:20 2011 (by Create and Import Peripheral
##############################################################################

lib proc_common_v3_00_a all vhdl
lib plbv46_slave_single_v1_01_a all vhdl
lib receiver_v1_00_a STARTER vhdl
lib receiver_v1_00_a RECEIVER_FSM vhdl
lib receiver_v1_00_a RECEIVER vhdl
lib receiver_v1_00_a CLK_GENERATOR vhdl
lib receiver_v1_00_a CAN_SYSTEM vhdl
lib receiver_v1_00_a user_logic vhdl
lib receiver_v1_00_a receiver vhdl
```

## 8.5. Fase de pruebas

La fase de pruebas del sistema embebido es bastante escasa puesto que lo único que interesaba en este momento es conocer si todos los componentes del sistema embebido funcionaban correctamente.

Para ello se exportó el diseño hardware a la herramienta de desarrollo de software SDK para generar una aplicación de prueba de todos los periféricos conectados al Microblaze. Esta aplicación es generada automáticamente por el SDK y fue descargada en la placa de evaluación para probar que cada componente funcionaba.

# Capítulo 9

# Desarrollo del software de control para el sistema embebido basado en el procesador Microblaze

## 9.1. Introducción

En este capítulo se expondrá todo el proceso de desarrollo del software de control del sistema embebido basado en el microprocesador software Microblaze.

El software se tendrá las siguientes funciones:

- Control de las entradas y salidas del sistema embebido.
- Control de los elementos de visualización con el LCD y los leds.
- Ejecutar la rutina de adquisición de datos del periférico receptor desarrollado en la primera fase del proyecto y conectado al sistema embebido

## 9.2. Análisis

En este apartado se analizará el sistema utilizando notación UML. Se desarrollará el modelo de casos de uso, el de datos y el de comportamiento.

### 9.2.1. Modelo de casos de uso

**Diagrama de casos de uso**

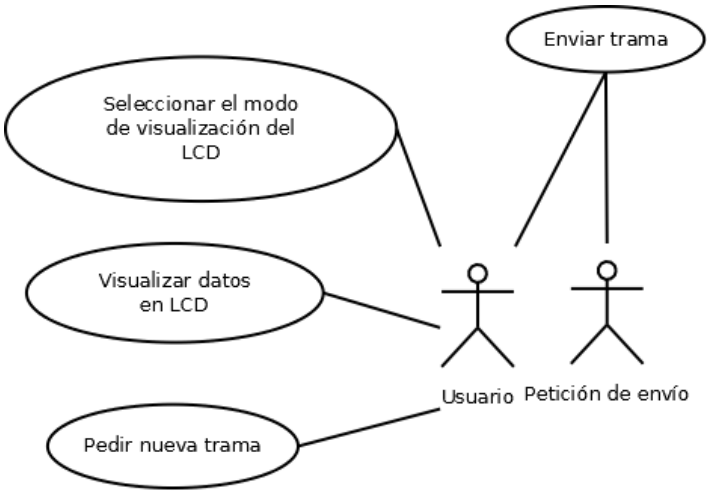


Figura 9.1: Casos de uso del sistema embebido

A continuación se describirán los distintos casos de uso del sistema con sus escenarios principales y alterna- tivos según la notación UML.

**Caso de uso: Seleccionar modo de visualización del LCD**

- **Caso de uso**: Seleccionar modo de visualización del LCD.
- **Identificación de escenarios**:
  - Escenario principal: El usuario selecciona el modo de visualización de datos en el LCD.
  - Escenario alternativo 1: El usuario selecciona el modo datos.
  - Escenario alternativo 2: El usuario selecciona el modo identificadores.
- **Descripción**: Selecciona los campos de la trama que serán visualizados en el LCD.
- **Actores**: Usuario.
- **Precondiciones**: Ninguna.
- **Postcondiciones**: El sistema cambia los campos de la trama a visualizar en el LCD.
- **Escenario principal**:
  1. El usuario quiere seleccionar el modo de visualización en el LCD.
  2. El usuario introduce el modo.
  3. El sistema comprueba el modo

4. El sistema actualiza los campos a visualizar.

- **Extensiones**:

4.a El modo es **"datos"**.

1. El sistema actualiza los campos del LCD con los datos de la trama recibida.

4.b El modo es **"identificadores"**.

1. El sistema actualiza los campos del LCD con los identificadores de la trama recibida.

**Caso de uso: Visualizar datos en el LCD**

- **Caso de uso**: Visualizar datos en el LCD.
- **Identificación de escenarios**:
  - Escenario principal: El usuario selecciona actualizar el LCD.
- **Descripción**: Actualiza el LCD con los campos de la trama en base al modo de visualización.
- **Actores**: Usuario.
- **Precondiciones**: Se ha seleccionado un modo de visualización.
- **Postcondiciones**: El sistema actualiza el LCD
- **Escenario principal**:
  1. El usuario quiere actualizar los datos del LCD.
  2. El usuario activa la visualización.
  3. El sistema prepara los datos con el formato a visualizar.
  4. El sistema actualiza el LCD.

**Caso de uso: Pedir nueva trama**

- **Caso de uso**: Pedir nueva trama.
- **Identificación de escenarios**:
  - Escenario principal: El usuario selecciona el modo pedir nueva trama.
  - Escenario alternativo 1: El usuario no selecciona pedir nueva trama.
- **Descripción**: Actualiza la trama a visualizar en la pantalla LCD.
- **Actores**: Usuario.
- **Precondiciones**: Ninguna.

- **Postcondiciones**: El registra la trama presente en el buffer del receptor para ser visualizada en la pantalla LCD.

- **Escenario principal**:

    1. El usuario desea pedir una nueva trama al receptor.
    2. El usuario selecciona la opción de **"nueva trama"**.
    3. El sistema comprueba el modo
    4. El sistema registra la trama.

- **Extensiones**:

    3.a No es el modo **"nueva trama"**.

        1. El sistema no registra la trama.

**Caso de uso: Enviar trama**

- **Caso de uso**: Enviar trama.

- **Identificación de escenarios**:

    - Escenario principal: El usuario selecciona el modo enviar trama y el sistema interactúa con la aplicación de monitorización externa.

- **Descripción**: Envía la trama presente en los registros del receptor del sistema a la aplicación externa.

- **Actores**: Usuario, petición de envío.

- **Precondiciones**: El usuario ha activado el modo de enviar trama.

- **Postcondiciones**: El sistema encapsula los datos de la trama en un buffer y lo envía a la aplicación externa.

- **Escenario principal**:

    1. El usuario activa el modo de transmisión.
    2. El sistema espera una petición de trama.
    3. El sistema calcula el número de bytes de los que se compone la transmisión en base al tipo de trama y la longitud de sus campos.
    4. El sistema envía el número de bytes a la aplicación externa.
    5. El sistema espera una petición de envío de la aplicación externa de monitorización.
    6. El sistema envía la trama a la aplicación externa.

### 9.2.2. Modelo conceptual de datos

El sistema está completamente orientado a objetos por lo que el siguiente modelo refleja completamente el sistema software que va a controlar el sistema embebido. La mayoría de las clases tienen como función el control los recursos de la placa de evaluación.

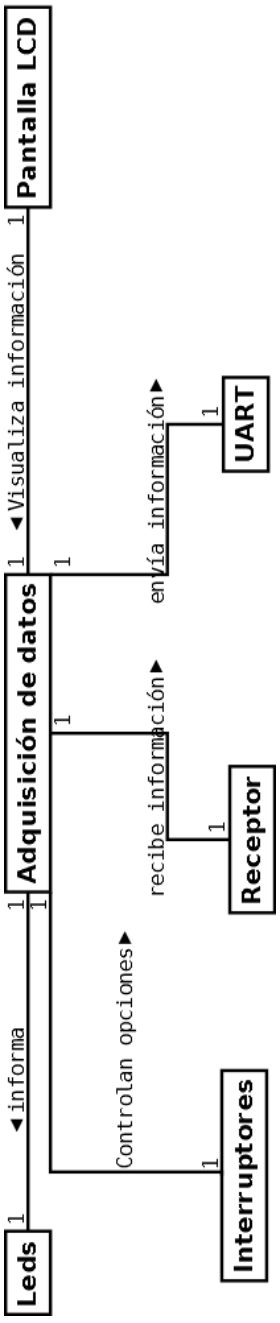


Figura 9.2: Modelo de datos del sistema embebido

### 9.2.3. Modelo de comportamiento del sistema

**Modelo de comportamiento de Seleccionar modo de visualización del LCD**

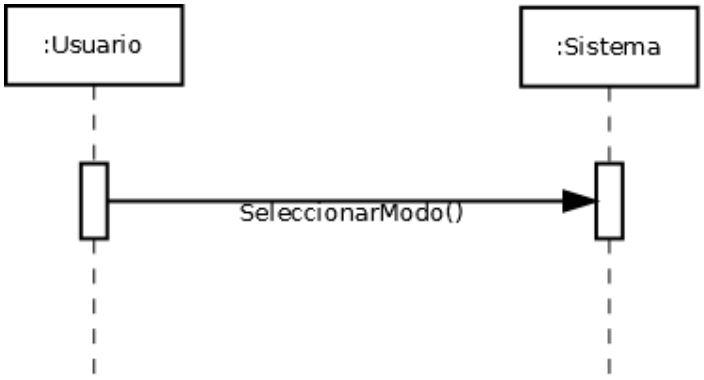


Figura 9.3: Diagrama de secuencia de seleccionar modo

**Contrato de las operaciones**

- **Operación: Seleccionar modo**
  - **Responsabilidades:** Selecciona entre los modos de visualización de los datos en la pantalla LCD.
  - **Referencias cruzadas:** Caso de uso Seleccionar modo de visualización del LCD.
  - **Precondiciones:** Existe un objeto I de la clase Interruptores.
  - **Postcondiciones:** El sistema cambia el modo en base al valor que devuelva I.

**Modelo de comportamiento de Visualizar datos en LCD**

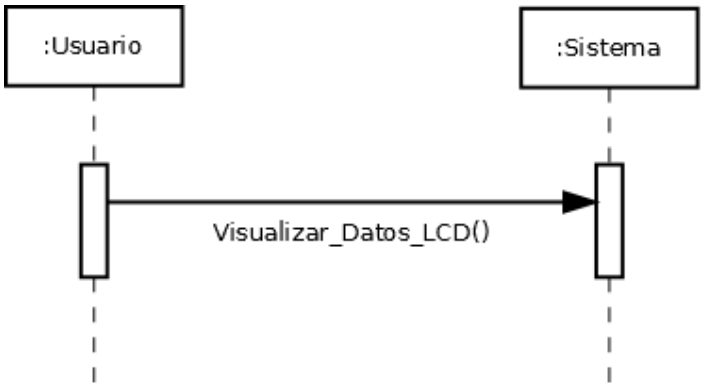


Figura 9.4: Diagrama de secuencia de visualizar datos en LCD

**Contrato de las operaciones**

- **Operación: Visualizar_datos_LCD()**

  - **Responsabilidades:** Actualiza los datos mostrados en la pantalla LCD.
  - **Referencias cruzadas:** Caso de uso Visualizar datos en LCD.
  - **Precondiciones:** Existe un L de la clase LCD y existe un objeto I de la clase Interruptores.
  - **Postcondiciones:** El sistema actualiza los datos a mostrar en L vuelve a bloquear L.

**Modelo de comportamiento de Pedir nueva trama**

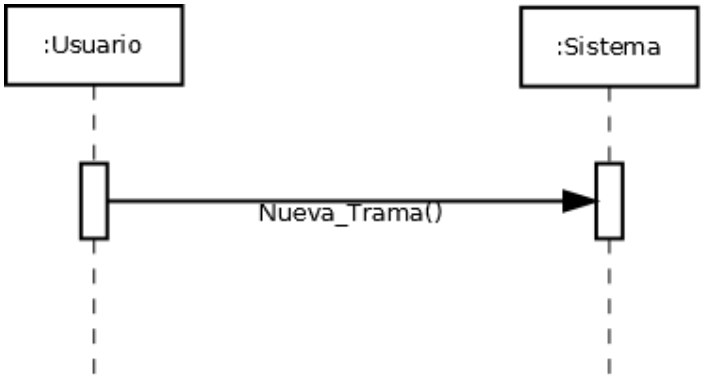


Figura 9.5: Diagrama de secuencia de pedir nueva trama

**Contrato de las operaciones**

- **Operación: Nueva_Trama()**

  - **Responsabilidades:** Obtiene una nueva trama del receptor para ser visualizada en la pantalla.
  - **Referencias cruzadas:** Caso de uso Pedir nueva trama.
  - **Precondiciones:** Existe un objeto I de la clase Interruptores y uno R de la clase Receptor.
  - **Postcondiciones:** El sistema obtiene una trama leyendo de los registros del receptor R y la almacena.

**Modelo de comportamiento de Enviar trama**

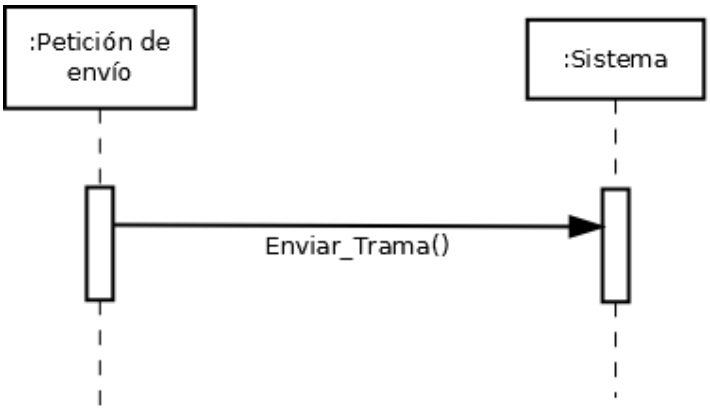


Figura 9.6: Diagrama de secuencia de enviar trama

**Contrato de las operaciones**

- **Operación: Enviar_Trama()**
  - **Responsabilidades:** Envía una trama completa a una aplicación externa.
  - **Referencias cruzadas:** Caso de uso Enviar trama.
  - **Precondiciones:** Existe un objeto R de la clase Receptor, existe un objeto U de la clase UART.
  - **Postcondiciones:** El sistema establece comunicación con la aplicación externa, prepara el buffer para la transmisión y U envía el buffer a la aplicación externa.

## 9.3. Diseño

Como en la sección de análisis, para el diseño también se seguirá la metodología orientada a objetos UML. Este proceso se facilita en gran medida una vez se ha realizado el análisis del mismo.

Es importante conocer que el diseño del sistema da una visión general del sistema que se implementará por lo que habrá determinados detalles del sistema final que se tienen en cuenta en la fase de implementación.

Además del diagrama de clases de diseño y la descripción de las operaciones de cada clase, está disponible toda la información que ha sido generada automáticamente a partir de los comentarios en el código con la herramienta Doxygen.

### 9.3.1. Diagrama de clases de diseño

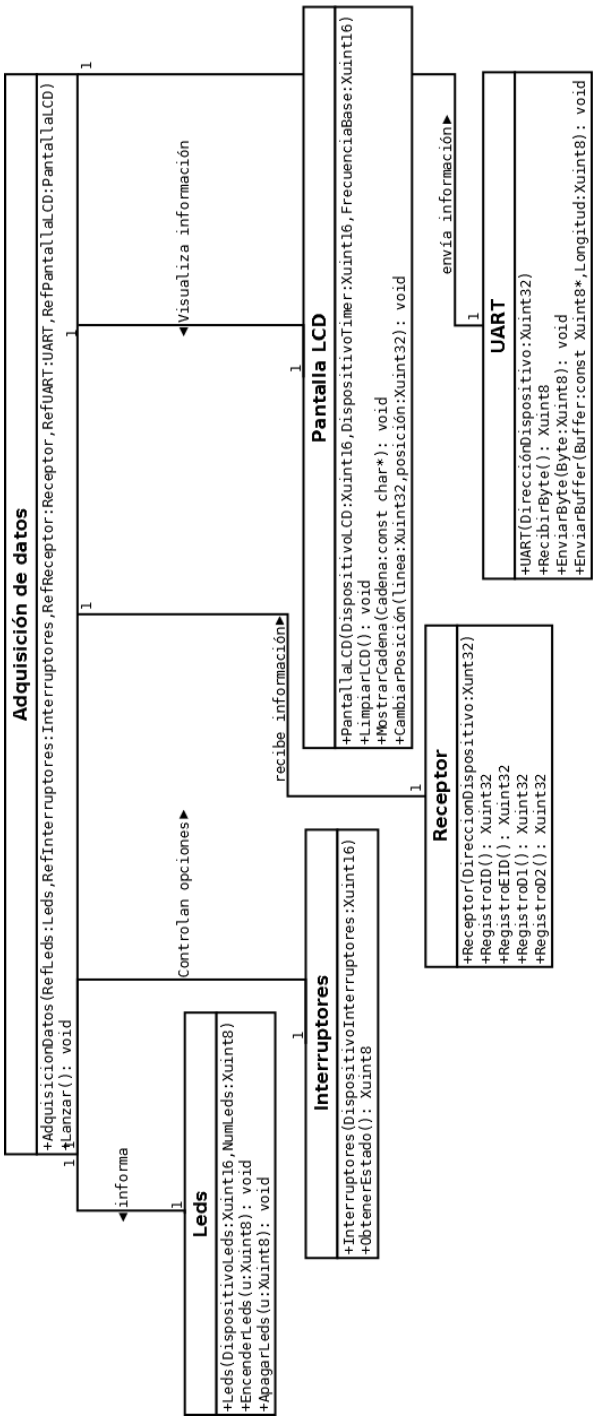


Figura 9.7: Diagrama de clases de diseño del software de control del sistema embebido

### 9.3.2. Comportamiento

A continuación se detallan todas las operaciones de cada clase así como los diagramas de secuencia obtenidos de la fase de diseño.

**Diagrama de secuencia de Seleccionar modo de visualización del LCD**

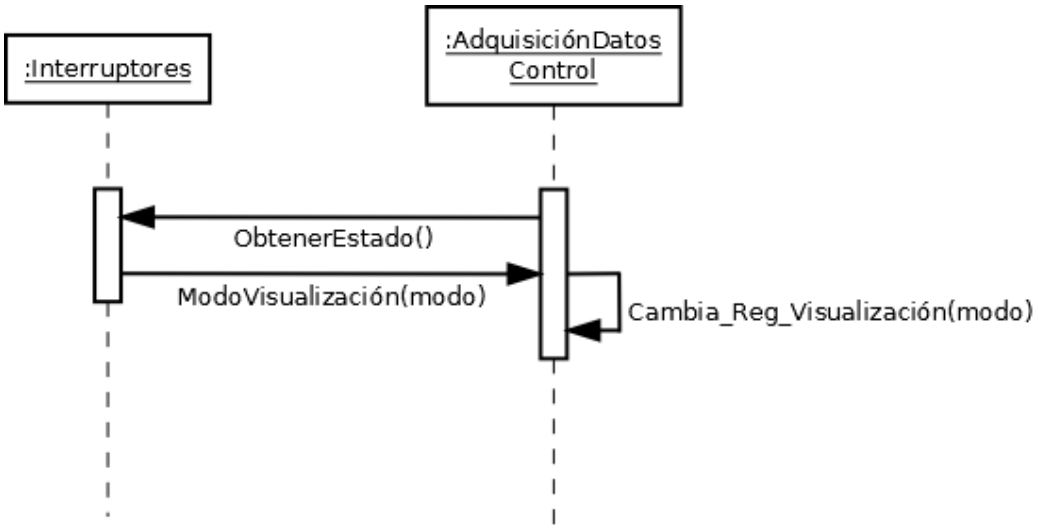


Figura 9.8: Diagrama de secuencia de Seleccionar modo de visualización del LCD

**Diagrama de secuencia de Visualizar datos en el LCD**

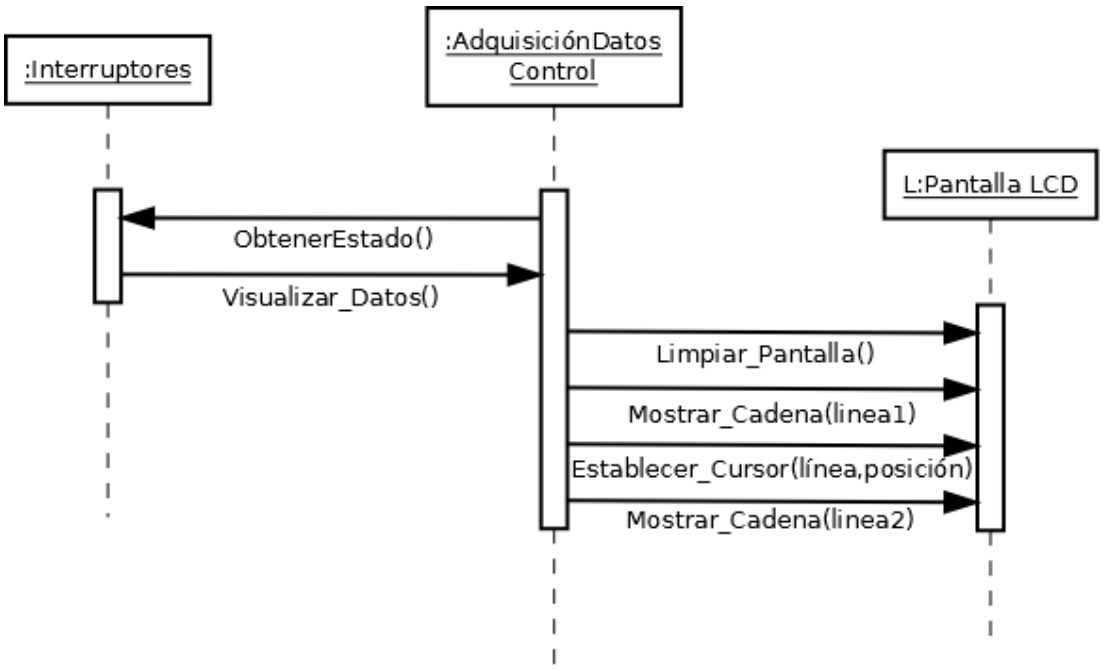


Figura 9.9: Diagrama de secuencia de Visualizar datos en el LCD

**Diagrama de secuencia de Pedir nueva trama**

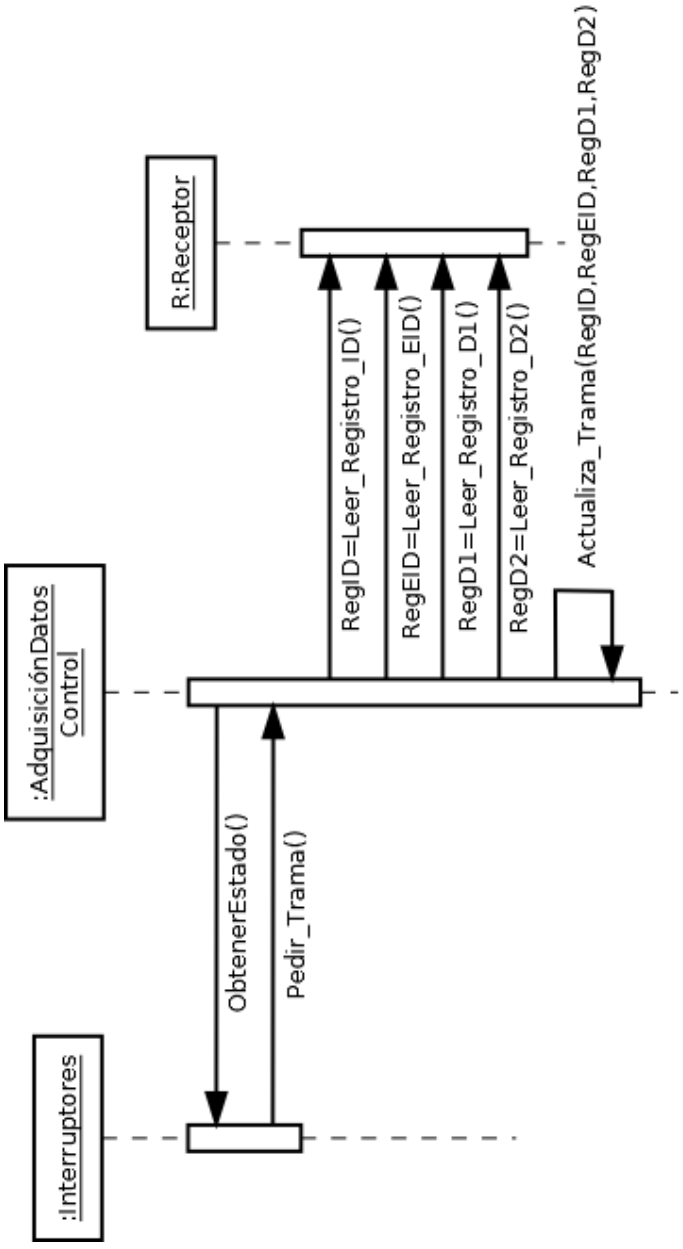


Figura 9.10: Diagrama de secuencia de Pedir nueva trama

**Diagrama de secuencia de Enviar trama**

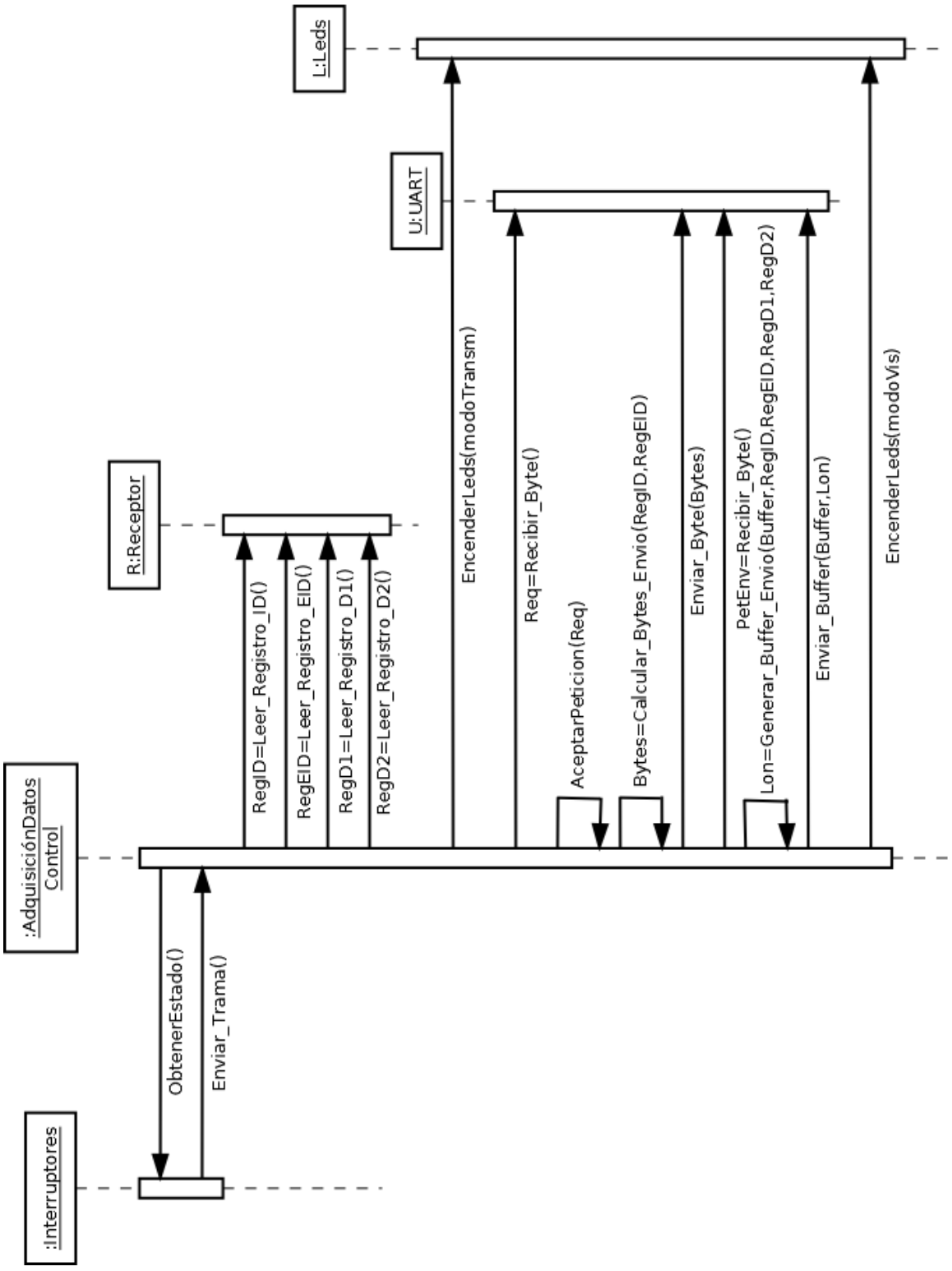


Figura 9.11: Diagrama de secuencia de Enviar trama

**Clase Leds**

- **Operación EncenderLeds(En_leds).** Enciende los leds indicados por el parámetro En_leds.
- **Operación ApagarLeds(Ap_leds).** Apaga los leds indicados por el parámetro Ap_leds.

**Clase Interruptores**

- **Operación ObtenerEstado().** Obtiene el estado de los interruptores.

**Clase Receptor**

- **Operación Leer_Registro_ID().** Obtiene el valor del registro del periférico donde está almacenado el identificador de las tramas recibidas.
- **Operación Leer_Registro_EID().** Obtiene el valor del registro del periférico donde está almacenado el identificador extendido de las tramas recibidas y los bits de estado.
- **Operación Leer_Registro_D1().** Obtiene el valor del registro del periférico donde están almacenados los 32 primeros bits las tramas recibidas.
- **Operación Leer_Registro_D2().** Obtiene el valor del registro del periférico donde están almacenados los 32 últimos bits las tramas recibidas.

**Clase UART**

- **Operación Recibir_Byte().** Espera la recepción de un byte por la UART del sistema y lo devuelve.
- **Operación Enviar_Byte(b).** Envía el byte "b" por la UART del sistema.
- **Operación Enviar_Buffer(buf,lon).** Invoca a **"Enviar_Byte(b)"** "lon" veces para enviar el buffer completo por la UART del sistema.

**Clase Pantalla LCD**

- **Operación Limpiar_Pantalla().** Limpia la pantalla LCD.
- **Operación Mostrar_Cadena(cad).** Muestra la cadena en la pantalla LCD.
- **Operación Establecer_Cursor(línea,posición.** Cambia el cursor de la pantalla LCD a la posición determinada por los parámetros línea y posición.

**Clase Adquisición de datos control**

- **Operación Calcular_Bytes_Envio(Idreg,Idereg).** Calcula el número de bytes que tendrá la transmisión de la información en base a los valores de los parámetros Idreg e Idereg, ambos registros del receptor que contienen información sobre el tamaño de la trama almacenada en los registros del receptor.

- **Operación Generar_Buffer_Envio(Buffer,Idreg,Idereg,d1reg,d2reg).** Genera el buffer de la transmisión en el parámetro Buffer en base al valor de los cuatro parámetros que representan la información obtenida de una trama almacenada en los registros del receptor(Idreg, Idereg, d1reg, d2reg).

- **Operación Cambia_Reg_Visualización(modo).** Cambia el modo de visualización del LCD en base al parámetro modo.

- **Operación Actualiza_Trama(id,ide,d1,d2).** Registra la trama presente en los parámetros id, ide, d1, d2 en las variables de visualización del LCD.

- **Operación ModoVisualización(modo).** Invoca a la operación Cambia_Reg_Visualización (modo).

- **Operación Visualizar_Datos().** Visualiza la última trama pedida en la pantalla LCD. Invoca a la operación Limpiar_Pantalla de la clase LCD.
  Luego invoca Mostrar_Cadena(linea1) para mostrar la primera línea de las dos de las que se compone la visualización en el LCD. Cambia a la segunda línea invocando a Establecer_Cursor(línea,posición) de la clase LCD. Por último invoca a Mostrar_Cadena(linea2) para mostrar la segunda línea de datos.

- **Operación Pedir_Trama().** Pide una nueva trama al receptor. Invoca a las operaciones Leer_Registro_ID(), Leer_Registro_EID(), Leer_Registro_D1(), Leer_Registro_D2() de la clase Receptor.

- **Operación Activar_ModoTransmisión().** Activa el modo de transmisión de datos.

- **Operación Enviar_Trama().** Envía una trama por la UART del sistema a la aplicación externa de monitorización. Invoca a las operaciones Recibir_Byte() de la clase UART para esperar y recibir el evento de transmisión.
  Luego invoca a Calcular_Bytes_Envio(Idreg,Idereg).
  Una vez calculados los bytes necesarios, invoca a Enviar_Byte(b) con el resultado de la anterior operación. Ahora espera un nuevo evento de envío para enviar la trama completa invocando a Recibir_Byte().
  Una vez recibido el evento, invoca a Generar_Buffer_Envio(Buf,Idreg,Idereg,d1reg,d2reg) y por último envía ese buffer invocando a Enviar_Buffer(buf,lon).

## 9.4. Implementación

Una vez acabadas las fases de análisis y diseño del sistema queda codificar el software cumpliendo todos los requisitos definidos en las anteriores dos fases.

El lenguaje elegido es C++ por su flexibilidad, potencia y por ser el lenguaje de programación que más he usado a lo largo de la carrera. Es muy adecuado para codificar software para sistemas embebidos debido a que soporta orientación a objetos y es un lenguaje que posee muchas características de bajo nivel.

Además de usar C++, ha sido necesario utilizar una serie de bibliotecas/cabeceras de Xilinx para el control de todos los periféricos del sistema mediante software. Estas cabeceras han sido las siguientes:

- **xbasic_types.h:** Incluye los tipos básicos definidos por Xilinx optimizados para los sistemas embebidos basados en Microblaze.
- **xstatus.h:** Tipos para controlar el estado de los periféricos del sistema.
- **xgpio.h:** Funciones para el control del periférico GPIO. Con el GPIO controlamos los leds, interruptores y pantalla LCD de la placa de evaluación.
- **xil_io.h:** Funciones básicas de entrada-salida para periféricos desarrollados por el usuario. En el caso de este proyecto servirá para hacer las lecturas de los registros de periférico receptor conectado al sistema embebido y que ha sido desarrollado en este proyecto final de carrera.
- **xuartlite_l.h:** Funciones para el control de la UART.
- **xtmrctr.h:** Funciones de control del periférico Timer que controla los retardos necesarios para hacer funcionar la pantalla LCD.

A continuación se van a exponer las cuestiones más relevantes de implementación:

### 9.4.1. Control de la pantalla LCD

Para el control de la pantalla LCD vamos a escribir en el registro del periférico GPIO que está conectado a los pines de la pantalla LCD. Además, el control de los tiempos de escritura y borrado de la pantalla serán controlados por el periférico Xps_Timer.

#### Mapeo de los pines del LCD

Para el control más cómodo de los pines del LCD, se definirán las siguientes macros:

```
#define LCD_DB4     0x01
#define LCD_DB5     0x02
#define LCD_DB6     0x04
#define LCD_DB7     0x08
#define LCD_RW      0x10
#define LCD_RS      0x20
#define LCD_E       0x40
```

Cada macro pone a 1 el bit correspondiente al pin que se desea controlar.

#### Envío de datos al LCD

Puesto que el LCD dispone de 4 líneas de datos para enviarle información y los datos a enviar (instrucciones y caracteres) son de 8 bits, son necesarias dos operaciones de escritura para enviar los datos completos. La primera escritura será la del nibble superior de los datos y la segunda la del inferior. Entre las dos escrituras es necesario esperar 100 $\mu$s con el usando el Timer.

#### Proceso de inicialización

La pantalla LCD requiere un proceso de inicialización para dejar lista la pantalla para ser usada. Los pasos son los siguientes:

1. Esperar 15 ms para que esté disponible para ser leída por la FPGA.
2. Enviar al LCD el valor 0x2 durante 12 ciclos de reloj del LCD. Esos doce ciclos se corresponden con 12 ms de tiempo que habrá que esperar usando el Timer.
3. Esperar otros 15 ms antes de configurar el LCD.
4. Enviar al LCD el valor 0x28 que se corresponde con el modo por defecto del LCD de la placa de evaluación Spartan 3E Starter Kit. Este modo configura la pantalla al modo de 4 bit, 60 Hz, fuentes de 5x8 y 2 líneas.
5. Enviar el valor 0x06 para configurar el LCD al modo de avance automático del cursor. A la hora de imprimir una cadena de caracteres, a cada carácter recibido para visualizar mueve el cursor una posición automáticamente.
6. Enviar al LCD el valor 0x0C para activar el LCD.

#### Borrado de la pantalla

Para borrar la pantalla LCD se sigue el siguiente proceso:

1. Escribir a cero la señal LCD_RS para poner el LCD en modo instrucción y esperar 1 $\mu$s con el Timer.
2. Escribir en el LCD la instrucción Clear(0x01) y esperar 2 ms con el Timer.

### 9.4.2. Protocolo de comunicaciones usando la UART

Para poder comunicar el sistema con la aplicación externa de monitorización que será desarrollada en el siguiente capítulo es necesario definir un protocolo de comunicaciones software para que sea usado por ambas partes.

El primer requisito que se establece es que la UART en ambas partes sea configurada sin control de flujo. Hay dos motivos:

- Tener el completo control en las comunicaciones.
- Asegurar que las comunicaciones del sistema sea compatible con todo tipo de UART. Hay determinadas placas de evaluación que por ahorrar costes solo implementan los pines Rx y Tx para conectar a la UART de la que disponga la placa de evaluación. Como únicamente disponen de esos dos pines no soportan el control de flujo por software ni por hardware. Con el protocolo que se va a desarrollar será compatible con cualquier dispositivo.

**Protocolo**

El protocolo se basa en dos peticiones de información y dos envíos de información:

1. Solicitud de trama. La aplicación de monitorización que desea pedir una trama envía el carácter 'R' por la UART mientras que la aplicación transmisora (sistema embebido) espera la recepción del mismo.
2. Envío del número de bytes de la transmisión. Una vez recibida la solicitud de trama, el sistema embebido calcula el número de bytes necesarios para la transmisión y los envía por la UART.
3. Solicitud de envío. La aplicación que pidió la trama recibe el número de bytes calculados por el sistema embebido y ajusta su buffer de recepción. Una vez hecho esto manda una petición de envío (carácter 'S') para que el sistema embebido envíe la trama.
4. Envío de la trama completa. El sistema embebido recibe la petición y envía la trama por la UART.

En el siguiente diagrama podemos observar los pasos en los que consiste el protocolo y la dirección de los mensajes.

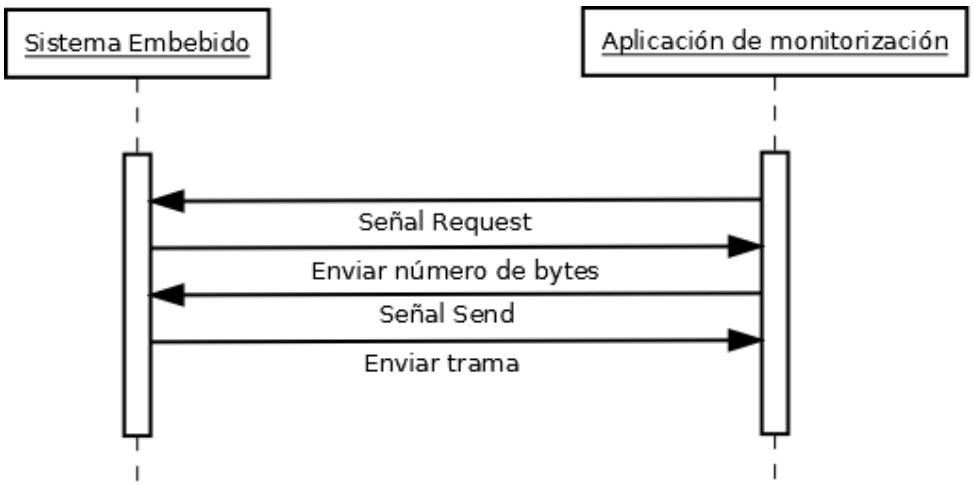


Figura 9.12: Protocolo de comunicaciones entre el sistema embebido y la aplicación de monitorización

### Información enviada

La siguiente tabla recoge la información que será enviada/recibida por el sistema embebido y la aplicación de monitorización.

| Señal/Envío | Datos | Bytes |
|---|---|---|
| Señal “Request” | Carácter ’R’ | 1. |
| Envío de número de bytes | Número entre 2 y 13 | 1. |
| Señal “Send” | Carácter ’S’ | 1. |
| Envío de trama | Datos de la trama | de 2 a 13. |

Tabla 9.1: Datos enviados por el protocolo de comunicaciones

### Formato del buffer para enviar la trama

En la siguientes dos tablas quedan recogidos los bits de datos que se enviarán según la trama sea estándar o de identificador extendido.

| Campo | Bits | Descripción |
|---|---|---|
| Identificador | 11 | Identificador de la trama.<br>1 Byte completo y 3 bits del segundo byte. |
| Trama con Id extendido | 1 | Bandera de Id extendido.<br>Vale 0. Bit 4 del segundo byte. |
| Longitud del campo de datos | 4 | 4 bits restantes del segundo byte. |
| Campo de datos de la trama | 0-64 | Se rellenan los bytes del 3 al 10 según la longitud del campo. |

Tabla 9.2: Buffer de envío de la trama estándar

| Campo | Bits | Descripción |
|---|---|---|
| Identificador | 11 | Identificador de la trama.<br>1 Byte completo y 3 bits del segundo byte. |
| Trama con Id extendido | 1 | Bandera de Id extendido.<br>Vale 1. Bit 4 del segundo byte. |
| Longitud del campo de datos | 4 | 4 bits restantes del segundo byte. |
| Identificador extendido | 18 | 2 bytes completos (3 y 4) y los 2 bits más significativos del quinto. |
| Relleno | 6 | Rellena el quinto byte. No usar estos bits. |
| Campo de datos de la trama | 0-64 | Se rellenan los bytes del 6 al 13 según la longitud del campo. |

Tabla 9.3: Buffer de envío de la trama con identificador extendido

## 9.5. Pruebas

La fase de pruebas del software es una de las más importantes dentro del ciclo de desarrollo del software debido a que permite la identificación de fallos en la implementación, en el funcionamiento, en el rendimiento y permite asegurar la calidad del código final.

### 9.5.1. Plan de pruebas

Al tratarse de software para un sistema embebido, el plan de pruebas sigue un esquema especial puesto el software debe ser descargado y probado en el hardware. Lo bueno del hardware en el que se va a ejecutar el software es que dispone de un módulo depurador hardware por lo que la depuración se realiza como si de un programa de computador se tratara simplemente teniendo conectado por USB la placa de evaluación al computador desde donde se depure.

El plan de pruebas desarrollado consiste en las siguientes fases:

- Pruebas de clases: Tras completar cada clase, la cual controla un componente hardware determinado, se probaban en el hardware observando que el comportamiento de todos sus método es el adecuado. Como las clases controlan componentes hardware, no hay posibilidad de casos extremos puesto que los componentes tienen un funcionamiento específico y muy bien determinado.
- Pruebas del sistema: Una vez están implementadas todas las clases que componen el sistema, se procede a la prueba de conjunto observando que el funcionamiento del sistema en conjunto es el esperado. Como en la anterior fase de pruebas, es necesario descargar el software en la placa de evaluación mediante la herramienta de desarrollo SDK de Xilinx.

Las pruebas han sido realizadas en dos momentos específicos durante el desarrollo de este software de control para el sistema embebido.

- Durante la implementación: La razón de probar durante el desarrollo es detectar a tiempo los errores de codificación que puedan surgir sobretodo a bajo nivel puesto que estos son los errores más difíciles de detectar y pueden afectar a los componentes hardware.
- Finalizada la fase de implementación: Se observa que el sistema funciona tal y como fue analizado y diseñado.

### 9.5.2. Diseño de las pruebas

- Durante la implementación: Para probar cada módulo de control de un componente hardware se probaron con datos adaptados a los requisitos de funcionamiento. A continuación se comenta las pruebas de cada componente:
    - Leds: Se probó a encender y apagar todos los leds para verificar el correcto funcionamiento de la clase de control de este componente hardware.

- Interruptores: Se probaron los interruptores obteniendo su estado, almacenándolo en una variable entera y observándolo con el depurador de la herramienta SDK.
- UART: Se probó a enviar y recibir bytes usando una terminal como "Hyperterminal" o "Putty" viendo los valores recibidos con el depurador del SDK y los enviados desde el mismo terminal.
- Receptor: Las pruebas consistieron en leer de los registros del receptor al que previamente se le había enviado una trama de prueba usando el "Generador de tramas" desarrollado también en este proyecto como un módulo VHDL. Este generador de tramas se descargó en una segunda placa de evaluación, la cual fue conectada al sistema embebido para que su receptor comenzase a recibir la trama de prueba.
- Pantalla LCD: Las verificaciones consistieron en escribir caracteres y cadenas de caracteres de prueba en la pantalla, en diversas posiciones de la misma. También se probó el borrado de la pantalla.

- Finalizada la implementación: Fueron probados todos los modos de funcionamiento del sistema. En el modo de transmisión de datos, como en este momento no se disponía de la aplicación de monitorización para ordenadores personales que fue desarrollada más tarde en este proyecto, se simuló su funcionamiento usando una terminal y conectando el sistema embebido y el ordenador de la terminal mediante conexión serial con el protocolo RS-232.

# Capítulo 10

# Desarrollo del software de monitorización

## 10.1. Introducción

En este capítulo se expondrá todo el proceso de desarrollo del software de monitorización para ordenadores personales.

El software se encargará de las siguientes tareas:

- Establecer comunicación con el sistema embebido de los dos capítulos anteriores.
- Realizar peticiones de tramas mediante el protocolo de comunicaciones definido en el capítulo anterior.
- Recibir las tramas del sistema embebido y visualizarlas mediante la interfaz de usuario.
- Guardar un log de toda la información recibida.

## 10.2. Análisis

En este apartado se analizará el sistema utilizando notación UML. Se desarrollará el modelo de casos de uso, el de datos y el de comportamiento.

### 10.2.1. Modelo de casos de uso

**Diagrama de casos de uso**

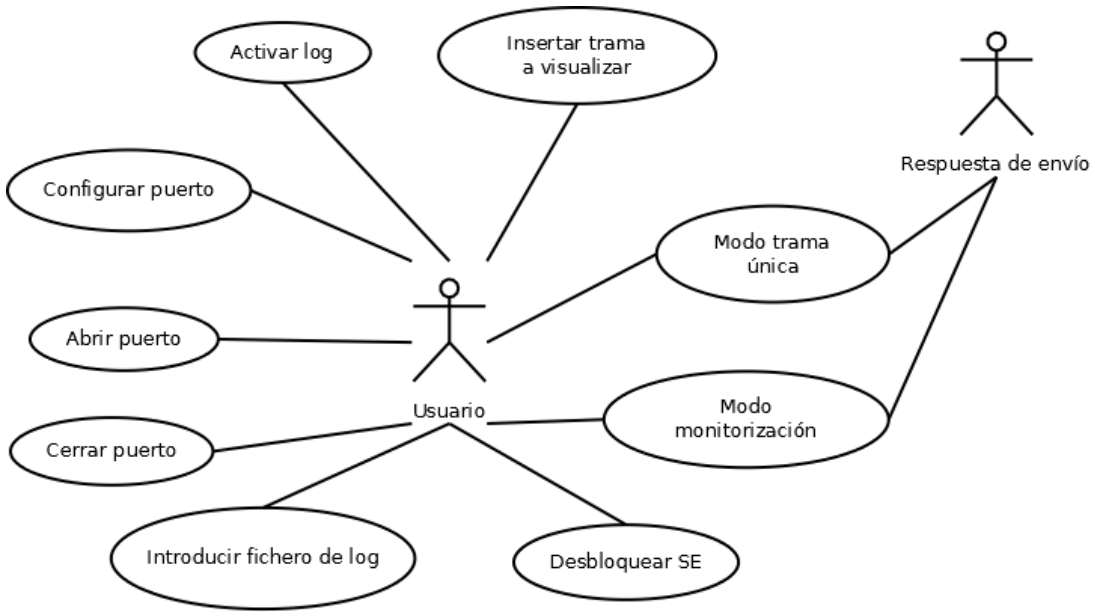


Figura 10.1: Casos de uso de la aplicación de monitorización

A continuación se describirán los distintos casos de uso del sistema con sus escenarios principales y alternativos según la notación UML.

**Caso de uso: Configurar puerto**

- **Caso de uso**: Configurar puerto.
- **Identificación de escenarios**:
  - Escenario principal: Se configura el puerto con todos los parámetros introducidos por el usuario.
  - Escenario excepción: El usuario cancela la configuración del puerto.
- **Descripción**: Configura las opciones del puerto serie.
- **Actores**: Usuario.
- **Precondiciones**: El puerto tiene que estar cerrado.
- **Postcondiciones**: El sistema registra la configuración del puerto.
- **Escenario principal**:
  1. El usuario quiere configurar el puerto serie.
  2. El sistema muestra las opciones a configurar.

3. El usuario introduce el puerto, la velocidad, la paridad y los bits de parada.
4. El usuario acepta los datos introducidos.
5. El sistema almacena la configuración del puerto.

- **Extensiones**:

  2-4.a El usuario cancela los datos introducidos.
      1. El sistema no almacena la configuración y borra la presente.

**Caso de uso: Abrir puerto**

- **Caso de uso**: Abrir puerto.
- **Identificación de escenarios**:
  - Escenario principal: Se abre el puerto correctamente.
  - Escenario error 1: Ya hay un puerto abierto en el sistema.
  - Escenario error 2: Configuración incorrecta para abrir el puerto.
- **Descripción**: Abre el puerto serie configurado en la aplicación.
- **Actores**: Usuario.
- **Precondiciones**: No hay ningún puerto abierto y existe una configuración válida.
- **Postcondiciones**: El puerto serie queda abierto
- **Escenario principal**:
  1. El usuario quiere abrir el puerto que ha configurado.
  2. El sistema comprueba que no hay un puerto abierto.
  3. El sistema recupera la configuración del puerto.
  4. El sistema abre el puerto con la configuración presente en la aplicación.
- **Extensiones**:

  2.a Hay un puerto abierto en el sistema.
      1. El sistema notifica que hay un puerto abierto.
      2. El sistema cancela la apertura del puerto.

  3.a No existe la configuración.
      1. El sistema notifica que la configuración presente no es válida.
      2. El sistema cancela la apertura del puerto.

**Caso de uso: Cerrar puerto**

- **Caso de uso**: Cerrar puerto.
- **Identificación de escenarios**:
  - Escenario principal: Se cierra el puerto correctamente.
  - Escenario error 1: No hay un puerto abierto en el sistema.
- **Descripción**: Cierra el puerto serie abierto por la aplicación.
- **Actores**: Usuario.
- **Precondiciones**: Hay un puerto abierto.
- **Postcondiciones**: El puerto serie queda cerrado.
- **Escenario principal**:
  1. El usuario quiere cerrar el puerto abierto por la aplicación.
  2. El sistema comprueba que hay un puerto abierto.
  3. El sistema cierra el puerto.
- **Extensiones**:

  2.a No hay un puerto abierto en el sistema.
    1. El sistema notifica que no hay un puerto abierto.
    2. El sistema cancela el cierre del puerto.

**Caso de uso: Desbloquear SE**

- **Caso de uso**: Desbloquear SE.
- **Identificación de escenarios**:
  - Escenario principal: Se desbloquea el sistema embebido correctamente.
  - Escenario de error: No hay un puerto abierto.
- **Descripción**: Desbloquea el modo de transmisión en el sistema embebido.
- **Actores**: Usuario.
- **Precondiciones**: Hay un puerto abierto.
- **Postcondiciones**: Se envía una señal de desbloqueo al sistema embebido.
- **Escenario principal**:
  1. El usuario quiere desbloquear el modo de transmisión del sistema embebido desde el software de monitorización.
  2. El sistema comprueba que hay un puerto abierto.

3. El envía la señal de desbloqueo.

- **Extensiones**:

  2.a No hay un puerto abierto en el sistema.
    1. El sistema notifica que no hay un puerto abierto.
    2. El sistema cancela el envío de la señal de desbloqueo.

**Caso de uso: Introducir fichero Log**

- **Caso de uso**: Introducir fichero Log.
- **Identificación de escenarios**:
  - Escenario principal: Se registra el camino del fichero correctamente.
  - Escenario excepción: El usuario cancela el proceso.
- **Descripción**: Pide el fichero donde se almacenará el log de recepción de información.
- **Actores**: Usuario.
- **Precondiciones**: Ninguna.
- **Postcondiciones**: El sistema almacena el camino al fichero donde se almacenará el log.
- **Escenario principal**:
  1. El usuario desea seleccionar un fichero para almacenar el log.
  2. El usuario introduce el fichero.
  3. El sistema registra el camino al fichero.
- **Extensiones**:

  *.a El usuario cancela el proceso.
    1. El sistema no almacena el camino al fichero.

**Caso de uso: Activar/Desactivar log**

- **Caso de uso**: Activar/Desactivar log.
- **Identificación de escenarios**:
  - Escenario principal: Se activa el modo log.
  - Escenario alternativo: Se desactiva el modo log.
- **Descripción**: Activa/desactiva el modo del log de datos recibidos.
- **Actores**: Usuario.
- **Precondiciones**: Ninguna.

- **Postcondiciones**: El sistema activa/desactiva el modo log.
- **Escenario principal**:
    1. El usuario desea configurar el modo log.
    2. El selecciona la opción de activar log.
    3. El sistema registra la activación del log.
- **Extensiones**:

    2.a El usuario selecciona la opción de desactivar log.
        1. El sistema registra la desactivación del log.

**Caso de uso: Insertar trama a visualizar**

- **Caso de uso**: Insertar trama a visualizar.
- **Identificación de escenarios**:
    - Escenario principal: El usuario introduce todos los datos de la trama a visualizar y la registra.
    - Escenario excepción: El usuario cancela .
- **Descripción**: Se registran los datos de una trama para ser visualizada por pantalla.
- **Actores**: Usuario.
- **Precondiciones**: Ninguna.
- **Postcondiciones**: El sistema registra los datos de la trama.
- **Escenario principal**:
    1. El usuario desea registrar un nuevo tipo de trama.
    2. El usuario introduce los datos de la trama (identificador, identificador extendido y descripción).
    3. El usuario selecciona la opción de registrar.
    4. El sistema registra los datos de la trama y lo asocia al visualizador LCD correspondiente.
- **Extensiones**:

    *.a El usuario aborta la introducción de la trama.
        1. El sistema no registra los datos de la trama.

**Caso de uso: Modo trama única**

- **Caso de uso**: Modo trama única.
- **Identificación de escenarios**:
  - Escenario principal: El sistema interactúa con el sistema embebido pidiéndole una nueva trama, la recibe y la visualiza por pantalla.
  - Escenario alternativo 1: El modo log está activo y el sistema almacena la trama recibida en el log.
  - Escenario error: La transmisión de datos se corta.
- **Descripción**: El sistema interactúa con el sistema embebido pidiendo una nueva trama. La visualiza y almacena si el modo log está activo.
- **Actores**: Usuario.
- **Precondiciones**: Hay un puerto configurado y abierto. Si está activo el log, requiere que se seleccione un fichero para almacenar los datos.
- **Postcondiciones**: El sistema opera con la trama recibida.
- **Escenario principal**:
  1. El usuario desea recibir una única trama del sistema embebido.
  2. El sistema envía una petición de envío al sistema embebido.
  3. El sistema recibe la respuesta del sistema embebido.
  4. El sistema ajusta su buffer de recepción al tamaño recibido.
  5. El sistema envía la petición de trama.
  6. El sistema espera la respuesta.
  7. El sistema registra la trama recibida del sistema embebido.
  8. El sistema comprueba si la trama recibida ha sido registrada por el usuario.
  9. El sistema comprueba si está activo el modo log.
- **Extensiones**:

  2-6.a Se corta la transmisión
  1. El sistema aborta las comunicaciones y para el modo monitor.

  7.a La trama fue registrada por el usuario
  1. El sistema la muestra en su visualizador correspondiente.

  7.b La trama no fue registrada por el usuario
  1. El sistema no la muestra.

  8.a Está activo el modo log.
  1. El sistema guarda la trama en el fichero de log registrado en el sistema por el usuario.

**Caso de uso: Modo monitorización**

- **Caso de uso**: Modo monitorización.
- **Identificación de escenarios**:
  - Escenario principal: El sistema recibe una trama del sistema embebido y la visualiza por pantalla. El proceso se repite mientras el usuario no lo pare.
  - Escenario alternativo 1: El modo log está activo y el sistema almacena la trama recibida en el log.
  - Escenario error: La transmisión de datos se corta.
- **Descripción**: El sistema interactúa con el sistema embebido pidiendo nuevas tramas. Las visualiza y almacena si el modo log está activo. El proceso no para hasta que el usuario lo indique.
- **Actores**: Usuario.
- **Precondiciones**: Hay un puerto configurado y abierto. Si está activo el log, requiere que se seleccione un fichero para almacenar los datos.
- **Postcondiciones**: El sistema opera con la trama recibida.
- **Escenario principal**:
  1. El usuario desea monitorizar tramas.
  2. El sistema envía una petición de envío al sistema embebido.
  3. El sistema recibe la respuesta del sistema embebido.
  4. El sistema ajusta su buffer de recepción al tamaño recibido.
  5. El sistema envía la petición de trama.
  6. El sistema espera la respuesta.
  7. El sistema registra la trama recibida del sistema embebido.
  8. El sistema comprueba si la trama recibida ha sido registrada por el usuario.
  9. El sistema comprueba si está activo el modo log.
  10. Repetir pasos 2 a 9 mientras el usuario no pare la monitorización.
- **Extensiones**:

  2-6.a Se corta la transmisión
    1. El sistema aborta las comunicaciones y para el modo monitor.

  8.a La trama fue registrada por el usuario
    1. El sistema la muestra en su visualizador correspondiente.

  8.b La trama no fue registrada por el usuario
    1. El sistema no la muestra.

9.a Está activo el modo log.

1. El sistema guarda la trama en el fichero de log registrado en el sistema por el usuario.

### 10.2.2. Modelo conceptual de datos

El sistema de monitorización está completamente orientado a objetos por lo que el siguiente modelo refleja completamente la aplicación de monitorización y los recursos del sistema que maneja como el puerto serie.

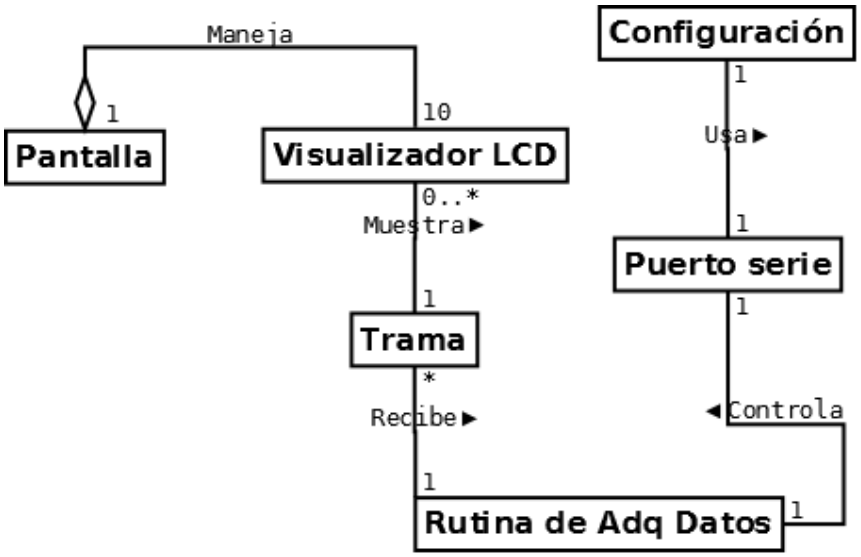


Figura 10.2: Modelo de datos de la aplicación de monitorización

### 10.2.3. Modelo de comportamiento del sistema

**Modelo de comportamiento de Configurar puerto**

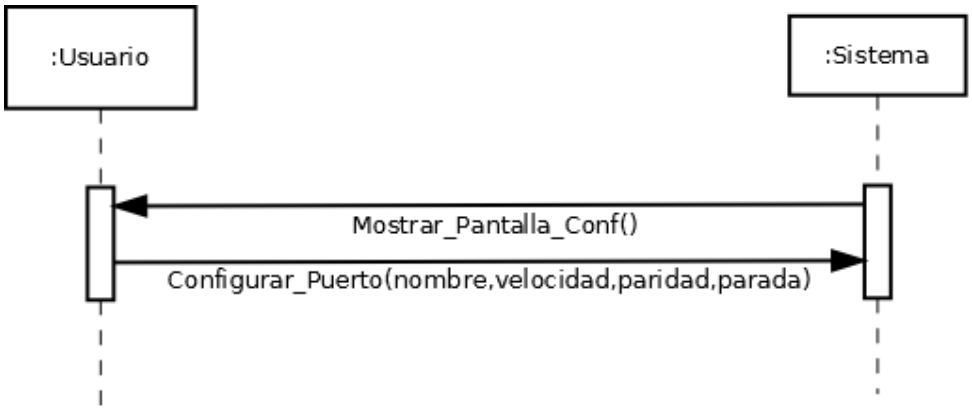


Figura 10.3: Diagrama de secuencia de Configurar puerto

**Contrato de las operaciones**

- **Operación: Configurar_puerto(nombre,velocidad,paridad,parada)**
  - **Responsabilidades:** Configura el puerto con las opciones pasadas como parámetros.
  - **Referencias cruzadas:** Caso de uso Configurar puerto.
  - **Precondiciones:** Ninguna.
  - **Postcondiciones:** Se crea una instancia C de configuración. Se asigna C.nombre=nombre, C.velocidad=velocidad, C.paridad=paridad y C.BitsParada=parada. Si existiese una configuración previa se eliminaría. Si se cancela la configuración del puerto se elimina la configuración actual.
- **Operación: Mostrar_Pantalla_Conf()**
  - **Responsabilidades:** Muestra el diálogo de configuración del puerto serie.
  - **Referencias cruzadas:** Caso de uso Configurar puerto.
  - **Precondiciones:** No se está ejecutando ningún modo de recepción de tramas.
  - **Postcondiciones:** El sistema muestra la pantalla de configuración del puerto serie.

**Modelo de comportamiento de Abrir puerto**


Figura 10.4: Diagrama de secuencia de Abrir Puerto

**Contrato de las operaciones**

- **Operación: Abrir_Puerto(Configuración)**
  - **Responsabilidades:** Abre el puerto con la configuración "Configuración".
  - **Referencias cruzadas:** Caso de uso Abrir puerto.
  - **Precondiciones:** No existe un P de Puerto serie tal que P.abierto=verdadero.
  - **Postcondiciones:** Se crea una instancia P de Puerto serie. Se asocia P con Configuración. Se abre el puerto P.

**Modelo de comportamiento de Cerrar puerto**

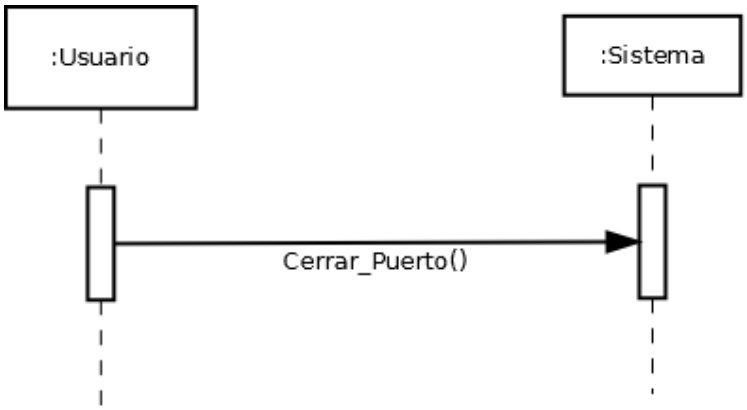


Figura 10.5: Diagrama de secuencia de Cerrar Puerto

**Contrato de las operaciones**

- **Operación: Cerrar_Puerto()**
  - **Responsabilidades:** Cierra el puerto serie abierto anteriormente por la aplicación.
  - **Referencias cruzadas:** Caso de uso Cerrar puerto.
  - **Precondiciones:** Existe un P de Puerto serie tal que P.abierto=verdadero y su configuración C asociada.
  - **Postcondiciones:** Se cierra el puerto P. Se elimina la asociación entre C de configuración y P de puerto serie. Se elimina la instancia P de Puerto serie.

**Modelo de comportamiento de Desbloquear SE**

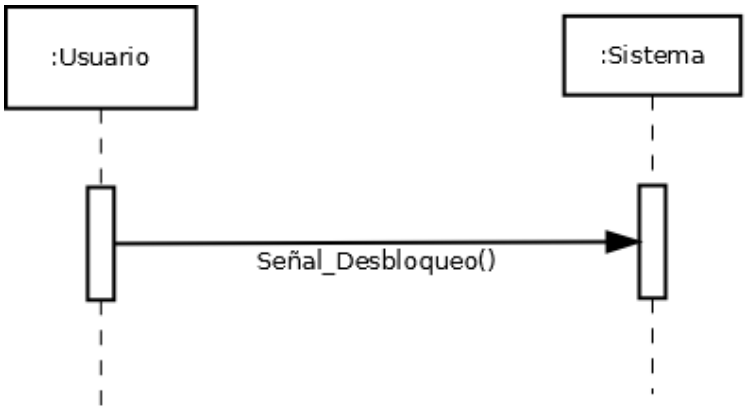


Figura 10.6: Diagrama de secuencia de Desbloquear SE

**Contrato de las operaciones**

- **Operación: Desbloquear_SE()**
  - **Responsabilidades:** Envía una señal de desbloqueo al sistema embebido.
  - **Referencias cruzadas:** Caso de uso Desbloquear SE.
  - **Precondiciones:** Existe un P de Puerto serie tal que P.abierto=verdadero.
  - **Postcondiciones:** Se envía una señal de desbloqueo por el puerto P.

**Modelo de comportamiento de Activar/Desactivar log**

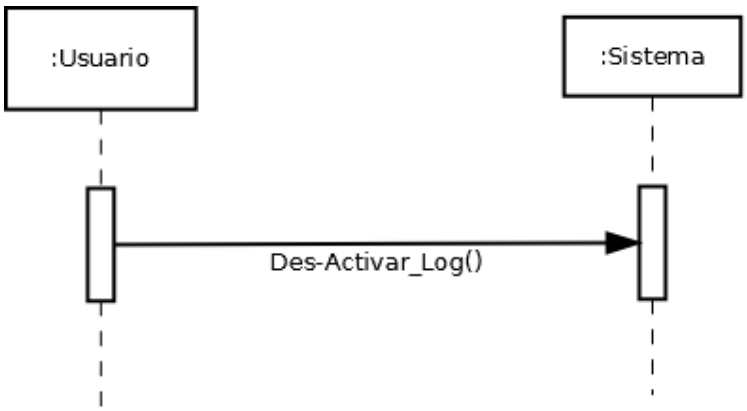


Figura 10.7: Diagrama de secuencia de Activar/Desactivar log

**Contrato de las operaciones**

- **Operación: Des-Activar_Log()**
  - **Responsabilidades:** Activa/desactiva la función de log de la aplicación.
  - **Referencias cruzadas:** Caso de uso Activar/Desactivar log.
  - **Precondiciones:** El sistema no está funcionando en uno de los dos modos de recepción de tramas.
  - **Postcondiciones:** Si está activa la función de log se desactiva y viceversa.

**Modelo de comportamiento de Introducir fichero Log**

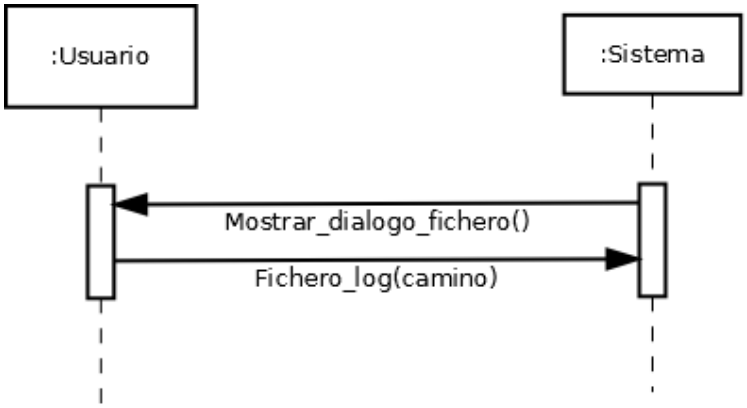


Figura 10.8: Diagrama de secuencia de Introducir fichero Log

**Contrato de las operaciones**

- **Operación: Mostrar_dialogo_fichero()**
  - **Responsabilidades:** Mostrar el diálogo para seleccionar un fichero para el log.
  - **Referencias cruzadas:** Caso de uso Introducir fichero Log.
  - **Precondiciones:** El sistema no está funcionando en uno de los dos modos de recepción de tramas.
  - **Postcondiciones:** El sistema muestra el diálogo para seleccionar un fichero para el log.
- **Operación: Fichero_log(camino)()**
  - **Responsabilidades:** Registrar el camino al fichero que almacenará el log de la recepción de tramas.
  - **Referencias cruzadas:** Caso de uso Introducir fichero Log.
  - **Precondiciones:** El sistema no está funcionando en uno de los dos modos de recepción de tramas.
  - **Postcondiciones:** El sistema registra el camino al fichero de log.

**Modelo de comportamiento de Insertar trama a visualizar**

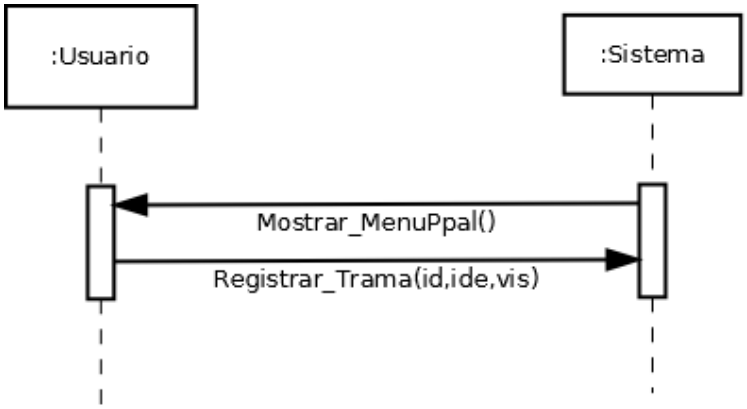


Figura 10.9: Diagrama de secuencia de Insertar trama a visualizar

**Contrato de las operaciones**

- **Operación: Registrar_Trama(id,ide,vis)**
  - **Responsabilidades:** Registrar una trama introducida por el usuario y asociarla a su visualizador.
  - **Referencias cruzadas:** Caso de uso Insertar trama a visualizar.
  - **Precondiciones:** Ninguna.
  - **Postcondiciones:** El sistema crea una instancia T de trama.
    Inicializa T con T.identificador=id, T.identificadorExt=ide.
    Se crea una asociación entre T y vis.

**Modelo de comportamiento de modo trama única**

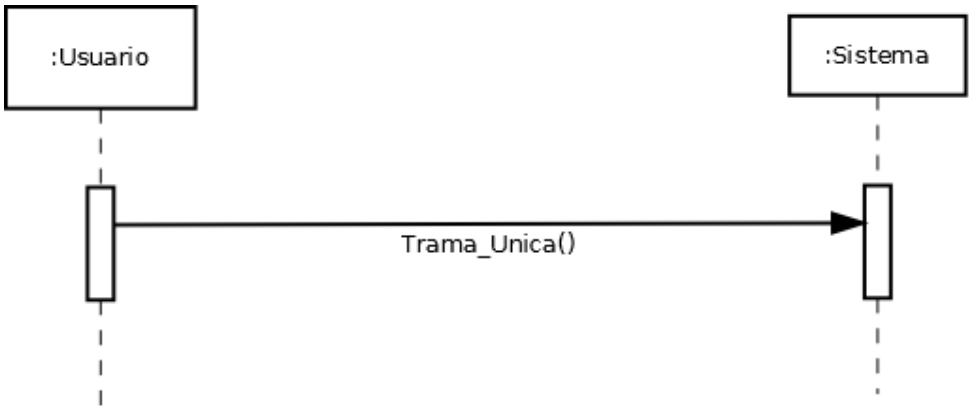


Figura 10.10: Diagrama de secuencia de Modo trama única

**Contrato de las operaciones**

- **Operación: Trama_Unica()**
  - **Responsabilidades:** Ejecutar el proceso de petición de una única trama al sistema embebido.
  - **Referencias cruzadas:** Caso de uso Modo trama única.
  - **Precondiciones:** Hay un puerto abierto en el sistema. Si se requiere funcionalidad de log debe haberse seleccionado un fichero de log.
  - **Postcondiciones:** El sistema realiza todo el proceso de petición y visualización de una trama recibida del sistema embebido.

**Modelo de comportamiento de modo monitorización**

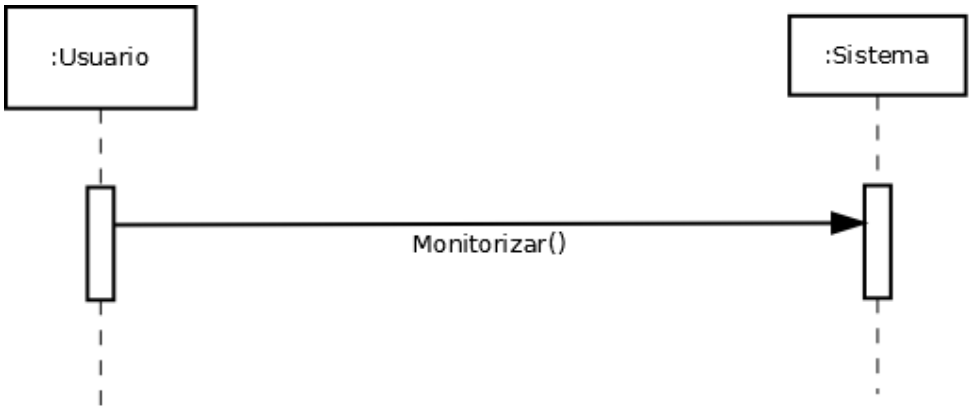


Figura 10.11: Diagrama de secuencia de Modo monitorización

**Contrato de las operaciones**

- **Operación: Monitorizar()**
  - **Responsabilidades:** Ejecutar el proceso de monitorización de tramas.
  - **Referencias cruzadas:** Caso de uso Modo monitorización.
  - **Precondiciones:** Hay un puerto abierto en el sistema. Si se requiere funcionalidad de log debe haberse seleccionado un fichero de log.
  - **Postcondiciones:** El sistema realiza todo el proceso monitorización recibiendo tramas del sistema embebido.

## 10.3. Diseño

Como en la sección de análisis, para el diseño también se seguirá la metodología orientada a objetos UML. Este proceso se facilita en gran medida una vez se ha realizado el análisis del mismo.

Es importante conocer que el diseño del sistema da una visión general del sistema que se implementará por lo que habrá determinados detalles del sistema final que se tienen en cuenta en la fase de implementación.

Además del diagrama de clases de diseño y la descripción de las operaciones de cada clase, está disponible toda la información que ha sido generada automáticamente a partir de los comentarios en el código con la herramienta Doxygen.

### 10.3.1. Diagrama de clases de diseño

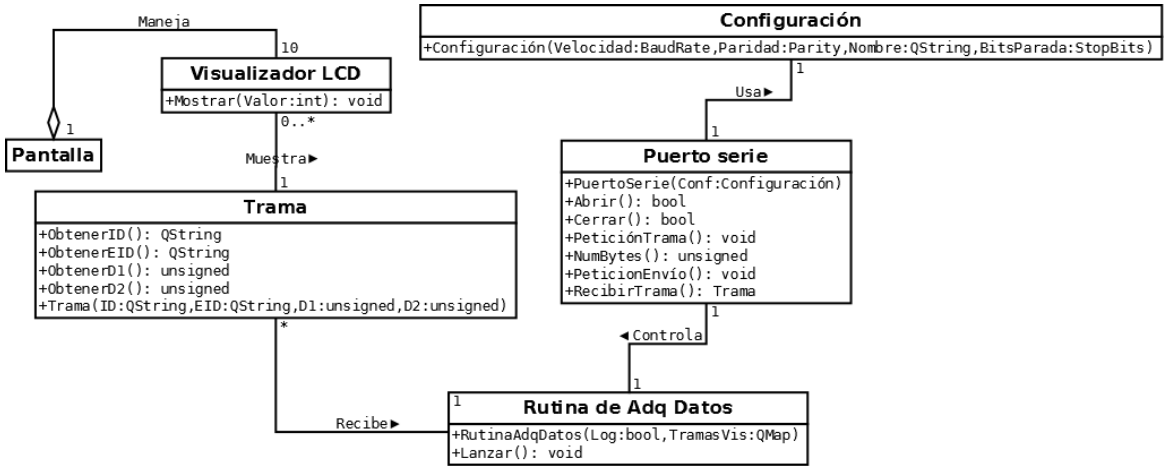


Figura 10.12: Diagrama de clases de diseño del software de monitorización

### 10.3.2. Comportamiento

A continuación se detallan todas las operaciones de cada clase así como los diagramas de secuencia obtenidos de la fase de diseño.

**Diagrama de secuencia de Configurar puerto**

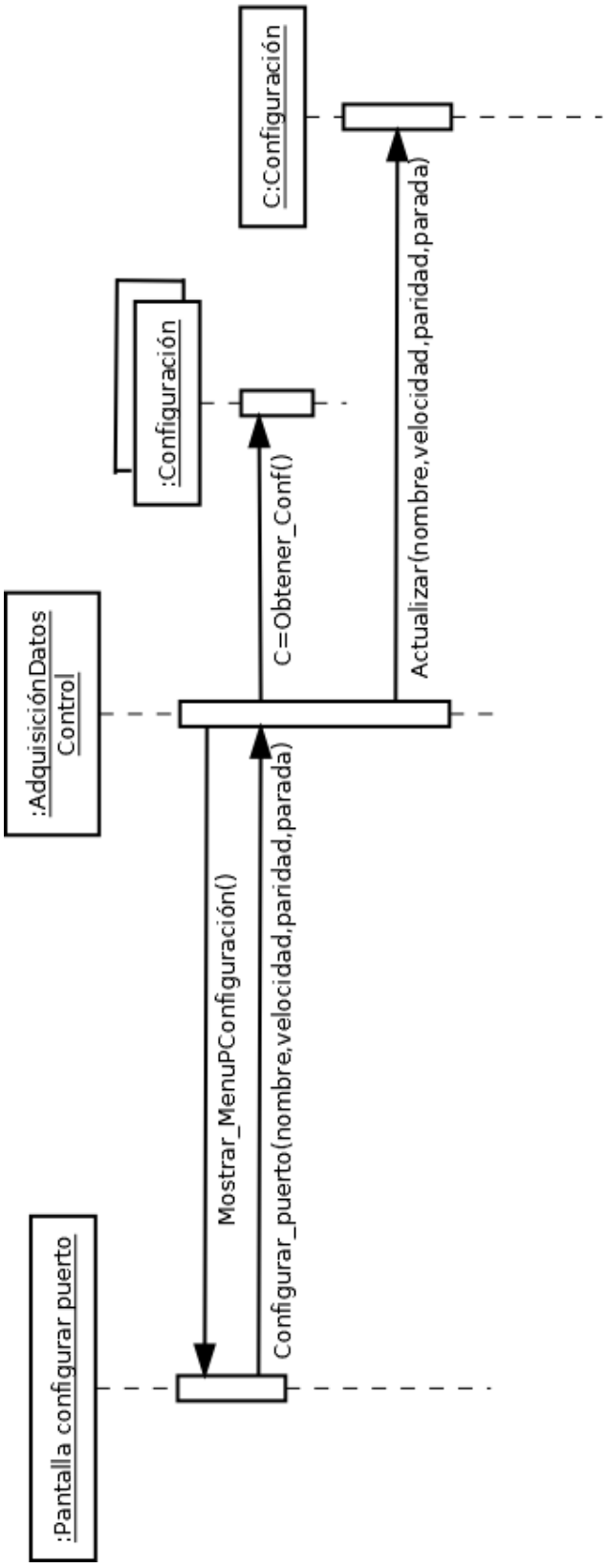


Figura 10.13: Diagrama de secuencia de Configurar puerto

**Diagrama de secuencia de Abrir puerto**

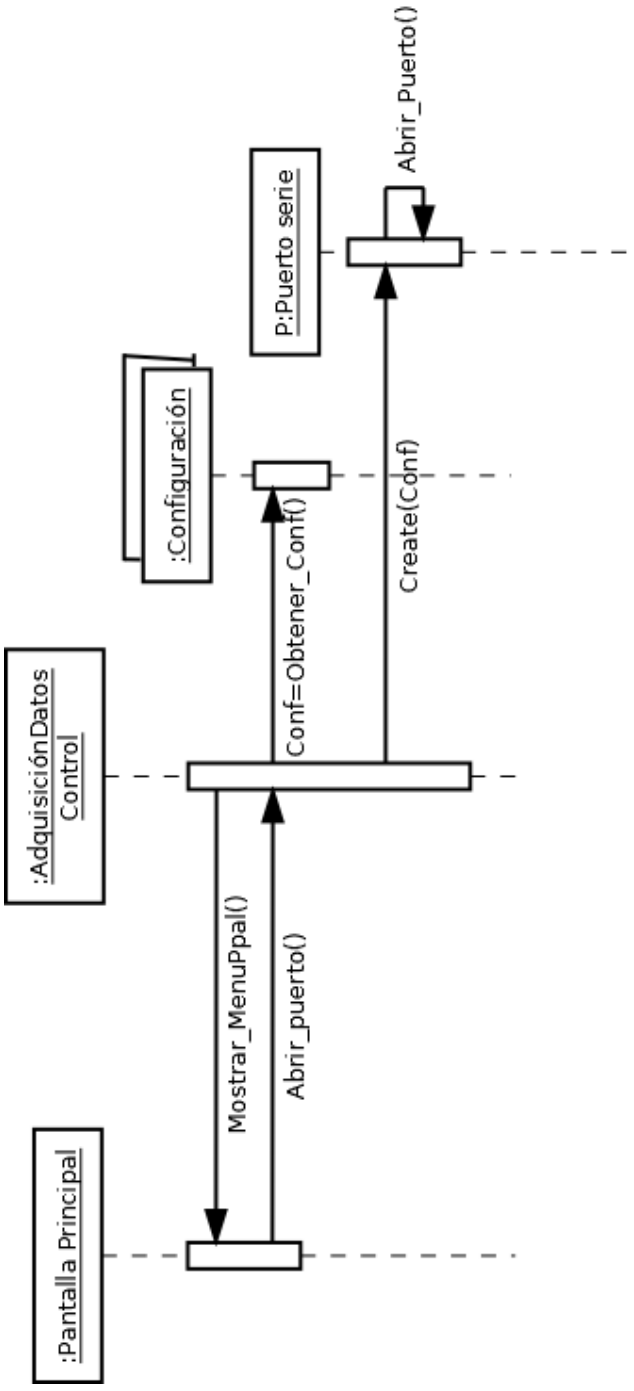


Figura 10.14: Diagrama de secuencia de Abrir puerto

**Diagrama de secuencia de Cerrar puerto**

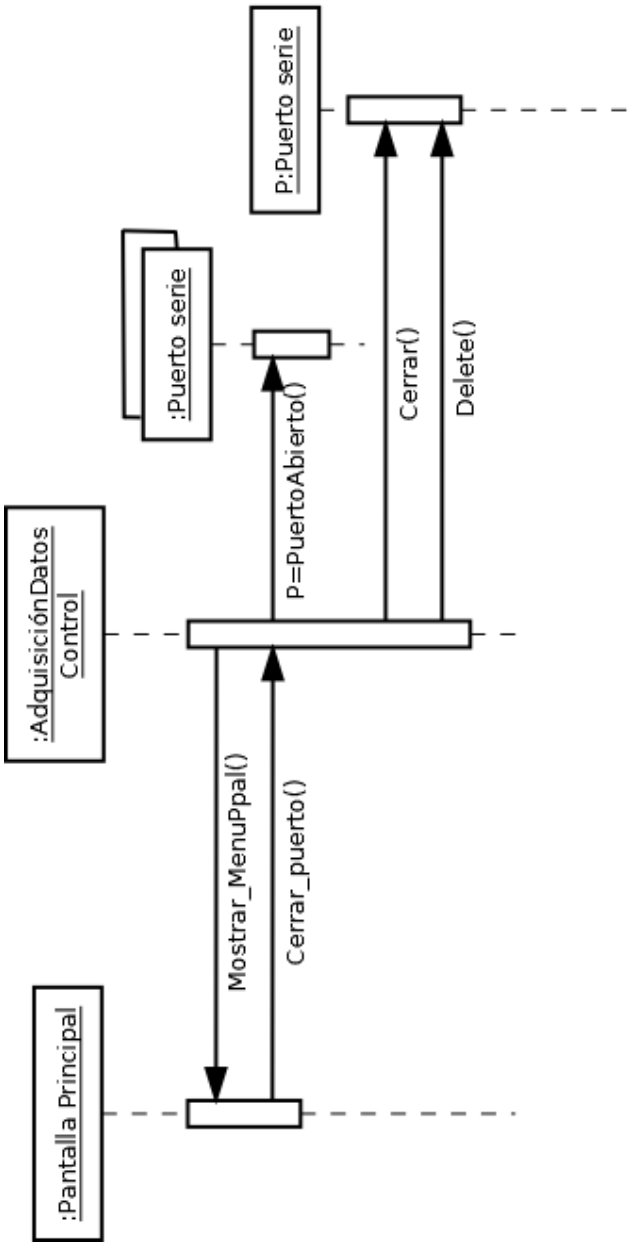


Figura 10.15: Diagrama de secuencia de Cerrar puerto

**Diagrama de secuencia de Activar/Desactivar log**

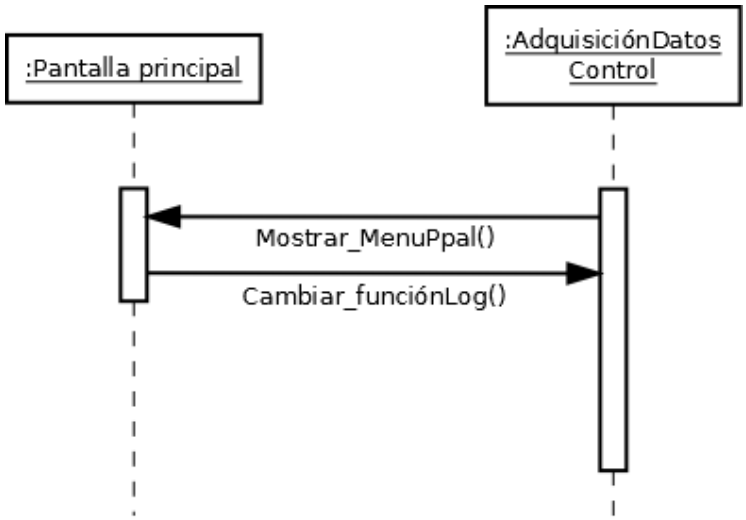


Figura 10.16: Diagrama de secuencia de Activar/Desactivar log

**Diagrama de secuencia de Introducir fichero Log**

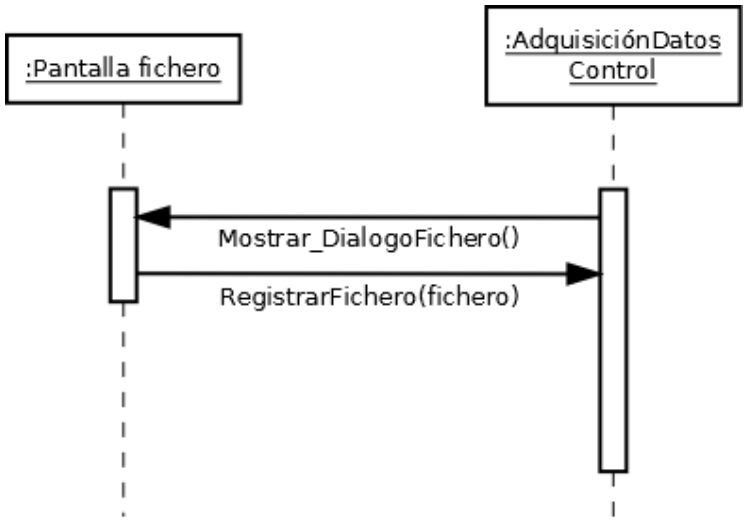


Figura 10.17: Diagrama de secuencia de Introducir fichero Log

**Diagrama de secuencia de Insertar trama a visualizar**

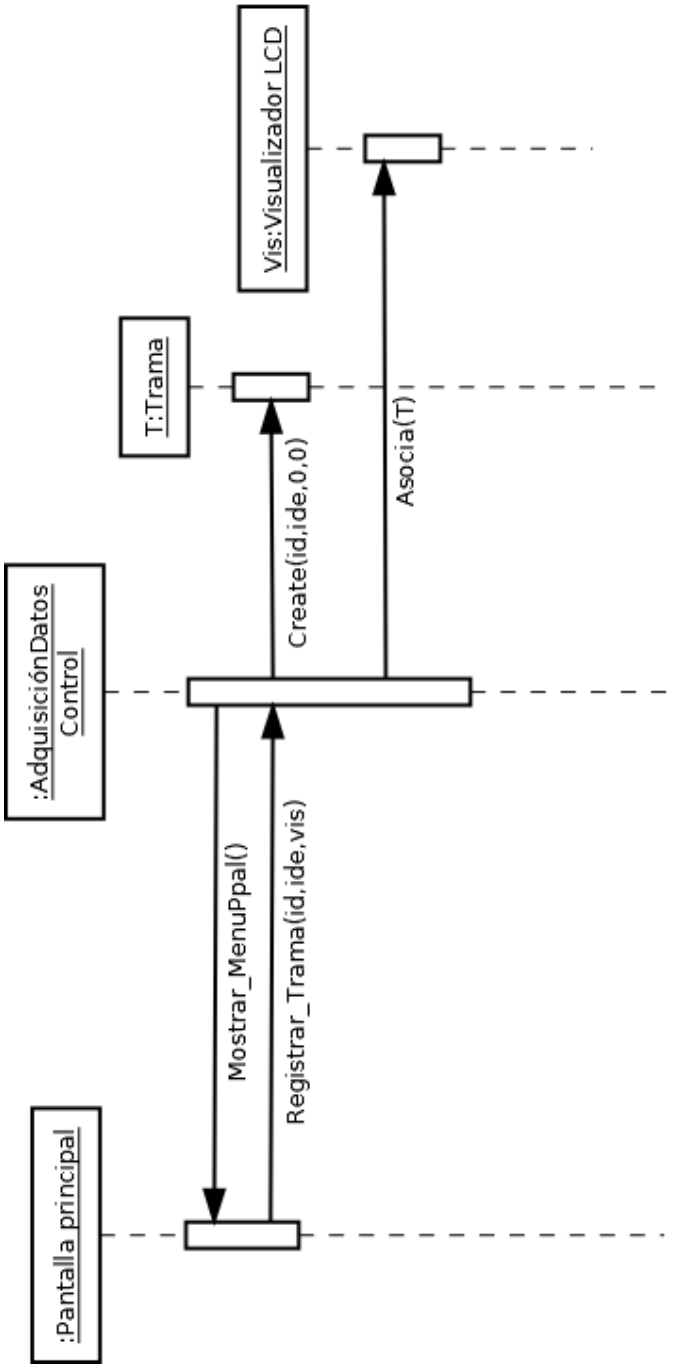


Figura 10.18: Diagrama de secuencia de Insertar trama a visualizar

**Diagrama de secuencia de Modo trama única**

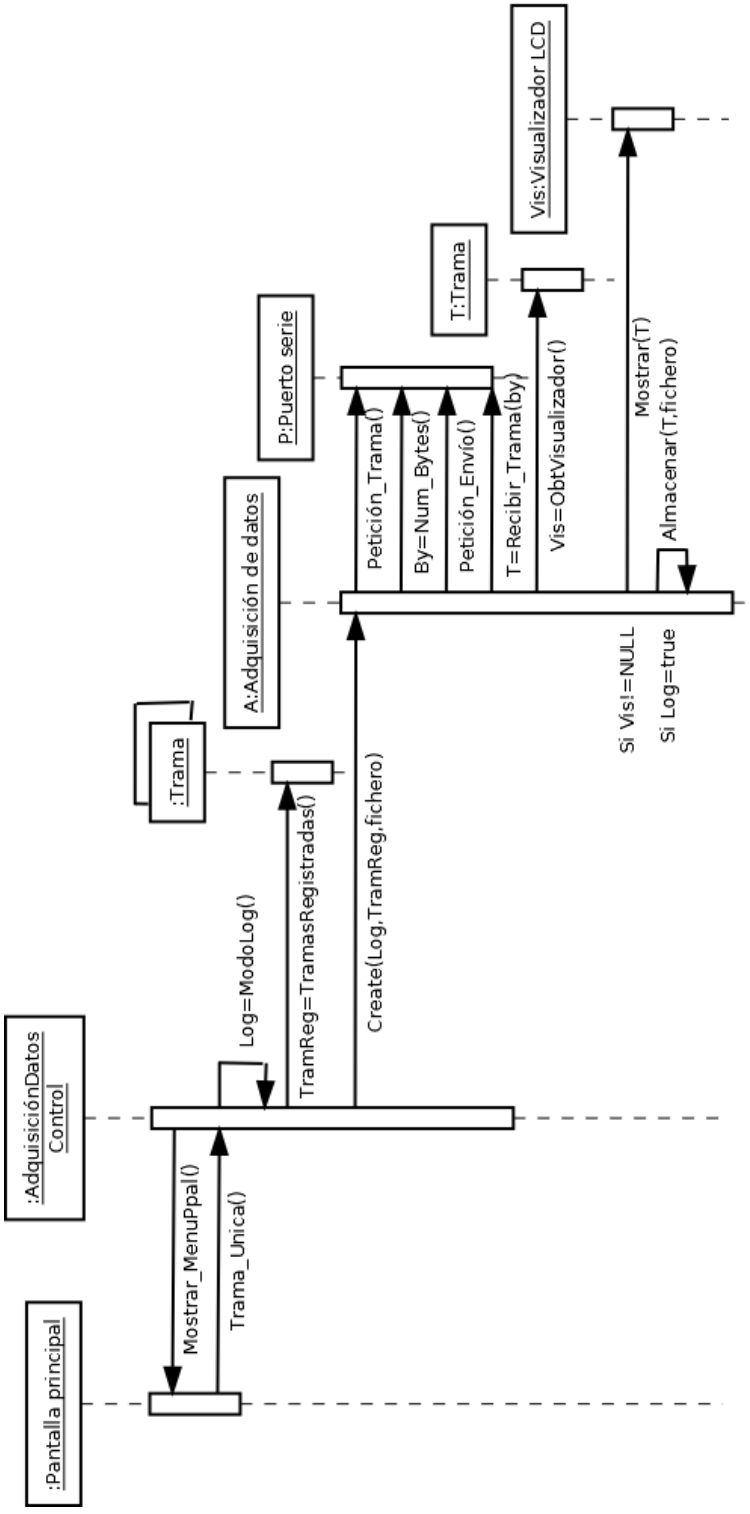


Figura 10.19: Diagrama de secuencia de Modo trama única

**Diagrama de secuencia de Modo monitorización**

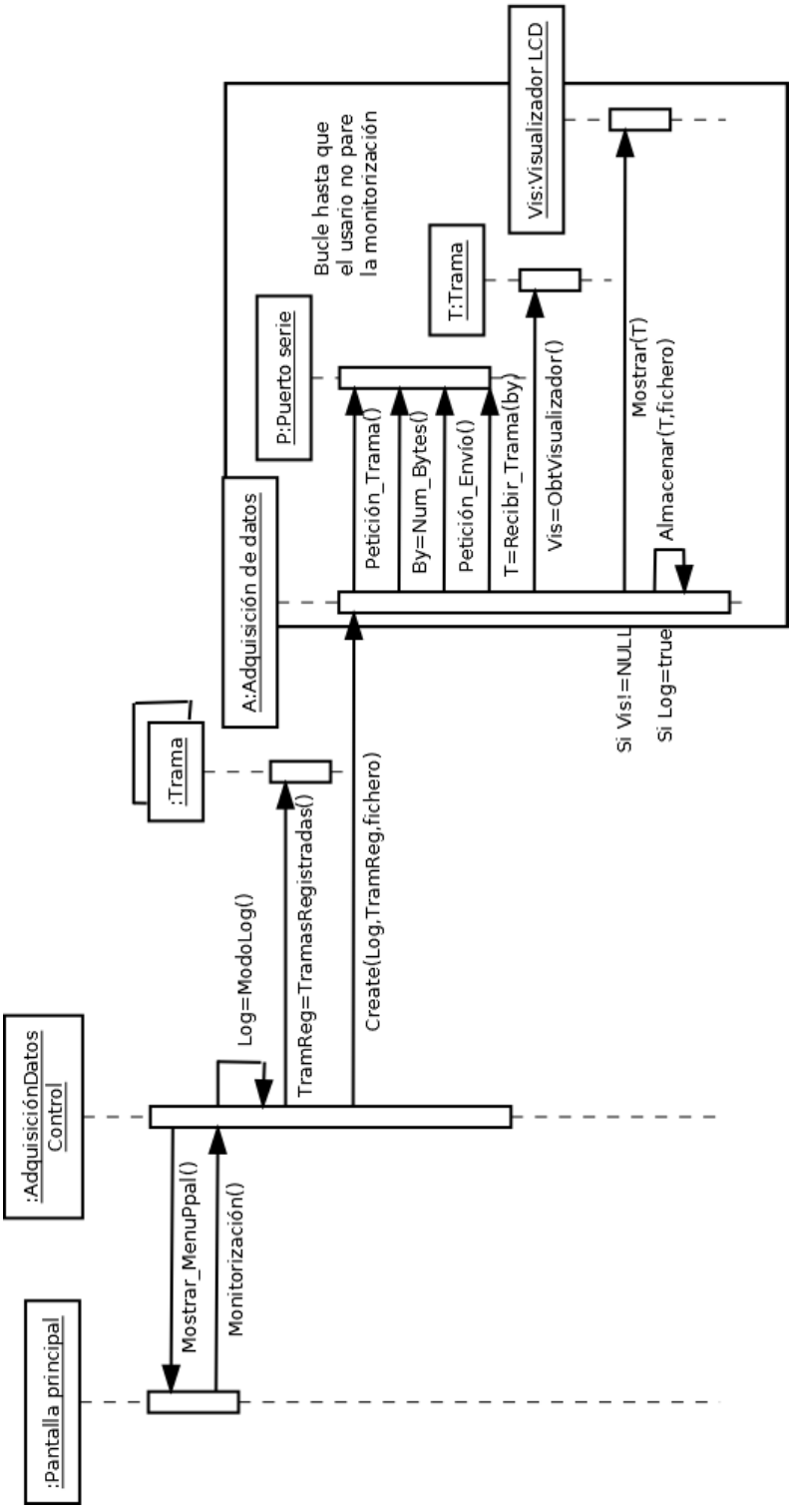


Figura 10.20: Diagrama de secuencia de Modo monitorización

**Diagrama de secuencia de Desbloquear SE**

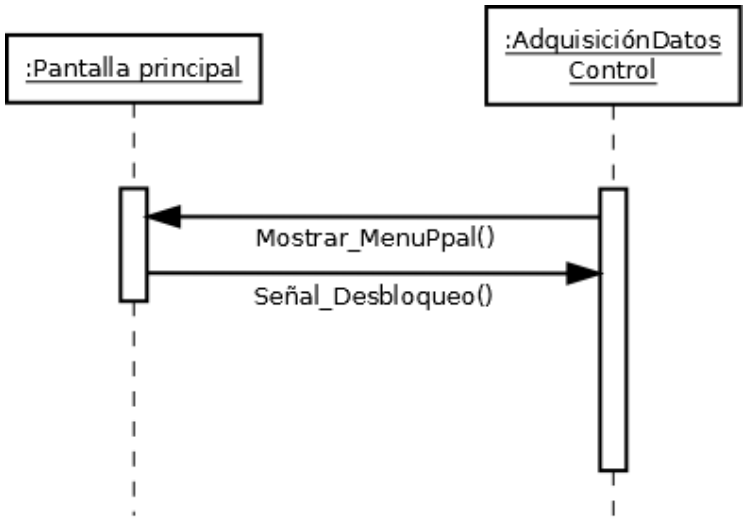


Figura 10.21: Diagrama de secuencia de Desbloquear SE

**Clase Pantalla Principal**

- **Operación Mostrar_MenuPpal().** Muestra el menú principal de la aplicación.

**Clase Pantalla fichero**

- **Operación Mostrar_DialogoFichero().** Muestra el dialogo para seleccionar un fichero de log.

**Clase Pantalla configurar puerto**

- **Operación Mostrar_MenuPConfiguración().** Muestra el menú de configuración del puerto serie.

**Clase Adquisición datos control**

- **Operación Cambiar_funciónLog().** Activa o desactiva el modo log dependiendo de la selección del usuario.
- **Operación Abrir_puerto().** Invoca a las operaciones AdquirirConf() de Configuración y Create(Conf) de Puerto serie.
- **Operación Configurar_puerto(nombre,velocidad,paridad,parada).** Invoca a las operaciones Obtener_Conf() y Actualizar(nombre,velocidad,paridad,parada) de Configuración.
- **Operación Cerrar_Puerto().** Invoca a las operaciones PuertoAbierto(), Cerrar() y Delete() de Puerto serie.

- **Operación RegistrarFichero(fichero).** Registra el camino al fichero de log “fichero” en el sistema.
- **Operación Registrar_Trama(id,ide,vis).** Invoca a las operaciones Create(id,ide,0,0) de la clase Trama y Asocia(T) de Visualizador LCD.
- **Operación Monitorización().** Invoca a las operaciones ModoLog() de la clase Adquisición datos control, TramasRegistradas() de Trama y Create(Log,TramReg,fichero) de Adquisición de datos.
- **Operación ModoLog().** Devuelve el si el modo Log está activo o no.
- **Operación Señal_Desbloqueo().** Envía una señal de desbloqueo al sistema embebido.

**Clase Adquisición de datos**

- **Operación Create(Log,TramReg,fichero).** Crea una instancia de Adquisición de datos con los parámetros introducidos por el usuario. Invoca a las operaciones Petición_Trama(), Num_Bytes(), Petición_Envío(), Recibir_Trama(bytes) de Puerto serie, ObtVisualizador() de Trama, Mostrar(T) de Visualizador LCD si el visualizador no es nulo y Almacenar(T,fichero) de Adquisición de datos.
- **Operación Almacenar(T,fichero).** Almacena la trama T en el fichero “fichero”.

**Clase Trama**

- **Operación Create(id,ide,0,0).** Crea una instancia de Trama con los identificadores introducidos por el usuario.
- **Operación TramasRegistradas().** Devuelve el conjunto de tramas registradas por el usuario.
- **Operación ObtVisualizador().** Devuelve el visualizador LCD asociado a la trama. Si no tiene devuelve NULL.

**Clase Visualizador LCD**

- **Operación Asocia(T).** Asocia el visualizador con la Trama T.
- **Operación Mostrar(T).** Muestra la trama T por el visualizador LCD.

**Clase Configuración**

- **Operación Obtener_Conf().** Devuelve la configuración del puerto serie presente en el sistema.
- **Operación Actualizar(nombre,velocidad,paridad,parada).** Actualiza la configuración con los parámetros pasados por el usuario.

**Clase Puerto serie**

- **Operación Create(Conf).** Crea una instancia de la clase Puerto serie con la configuración Conf e invoca a la operación Abrir_Puerto().
- **Operación Abrir_Puerto().** Abre el puerto serie que ha sido creado.
- **Operación PuertoAbierto().** Devuelve el puerto abierto en el sistema. Si no hay puerto abierto devuelve NULL.
- **Operación Cerrar_Puerto().** Cierra el puerto serie abierto del sistema.
- **Operación Delete().** Elimina la instancia de puerto serie presente en el sistema.
- **Operación Petición_Trama().** Manda una petición de trama nueva por el puerto serie del sistema.
- **Operación Num_Bytes().** Recibe el número de bytes de los que se va a componer la transmisión.
- **Operación Petición_Envío().** Manda una petición de envío por el puerto serie del sistema.
- **Operación Recibir_Trama(bytes).** Recibe una trama de longitud bytes por el puerto serie del sistema.

## 10.4. Implementación

Una vez acabadas las fases de análisis y diseño del sistema queda codificar el software cumpliendo todos los requisitos definidos en las anteriores dos fases. A continuación se van a exponer las cuestiones más relevantes de implementación:

El lenguaje elegido es C++ por su flexibilidad, potencia y por ser el lenguaje de programación que más he usado a lo largo de la carrera.

Puesto que la aplicación de monitorización que se va a implementar requiere de una interfaz de usuario amigable, es necesario elegir una biblioteca para interfaces gráficas. Hay varias opciones disponibles pero la elegida es la biblioteca Qt. En el siguiente apartado se describirán las características de esta biblioteca y los motivos de su elección.

### 10.4.1. Qt

Motivos de elección de Qt como biblioteca para la interfaz gráfica:

- Es multiplataforma: Todo el código escrito en Qt puede ser compilado y ejecutado en sistemas tan variados como Linux, Windows , Mac, etc...

- Ports casi instantáneos entre distintos sistemas operativos: El código de una aplicación desarrollada con Qt no necesita cambios entre distintos sistemas operativos lo que agiliza en gran medida el port entre sistemas. En el caso de la aplicación de monitorización desarrollada en este proyecto no ha sido necesario cambiar ni una sola línea de código para hacer que funcione tanto en Linux x86/x64 como en Windows x86/x64.
- Excelente documentación: Qt posee una excelente comunidad, una gran cantidad de plugins y bibliotecas para facilitar el uso de componentes tan variados de los sistemas como por ejemplo el puerto serie que ocupa un papel muy importante dentro de la aplicación.
- Su lenguaje nativo es C++.
- QtCreator: IDE de desarrollo de Nokia para aplicaciones Qt. Facilita el diseño e implementación de interfaces gráficas de forma completamente visual y además posee una gran base de datos de documentación y ejemplos de código.
- Es software libre.

### 10.4.2. Qextserialport

Para el control del puerto serie, es necesario utilizar una biblioteca externa a Qt puesto que Qt no posee capacidad para controlar el puerto serie. Para esta tarea se ha seleccionado la biblioteca Qextserialport que posee las siguientes características:

- Compatible con Qt.
- Compatible con sistemas Unix y Windows.
- Soporta velocidades propias de cada sistema operativo, modos de control de flujo por hardware/software y modos de funcionamiento por eventos y polling.
- Es software libre.

### 10.4.3. Protocolo de comunicaciones con el sistema embebido

Al igual que en el software de control del sistema embebido del capítulo anterior es necesario definir un protocolo de comunicaciones para comunicar el sistema embebido con la aplicación de monitorización. El protocolo es el mismo para ambas partes. Para consultarlo, ver el apartado 6.4.2 de este proyecto.

### 10.4.4. Formato del fichero de log

La capacidad de hacer un log de las tramas recibidas es una de las capacidades más interesantes de la aplicación de monitorización que se está desarrollando.

El formato del log consiste en una serie de campos que reflejan los datos más importantes de cada trama separados por dos puntos. Se ha decidido usar los dos puntos para separar los campos puesto que en caso de querer analizar el fichero de log se podrá usar una utilidad como *awk*

o *sed*.

Los campos de los que se compondrá el log son los siguientes:

| Campo | Descripción |
|---|---|
| Identificador | Identificador de la trama. |
| Id extendido | Identificador extendido de la trama. |
| Longitud del campo de datos | Número de bytes del campo de datos de la trama. |
| Descripción | Descripción de la trama introducida por el usuario. |
| Datos (bytes 1 a 4) | Campo de datos (4 primeros bytes de datos de la trama). |
| Datos (bytes 5 a 8) | Campo de datos (4 últimos bytes de datos de la trama). |
| Fecha | Fecha de recepción de la trama en formato –. |

Tabla 10.1: Campos del fichero de log

A continuación podemos ver un ejemplo de un Log capturado:

```
1881::4:Aceleracion:3474794979:0:mar jul 12 10:42:21 2011
1926:138780:1:Potencia:108:0:mar jul 12 10:42:21 2011
1881::5:Aceleracion:3474794979:108:mar jul 12 10:42:21 2011
1881::8:Aceleracion:3474794979:1827702588:mar jul 12 10:42:21 2011
1926:138780:1:Potencia:14:0:mar jul 12 10:42:21 2011
2015::2:Velocidad:54296:0:mar jul 12 10:42:21 2011
2015::4:Velocidad:3432583025:0:mar jul 12 10:42:22 2011
```

## 10.5. Pruebas

La fase de pruebas del software es una de las más importantes dentro del ciclo de desarrollo del software debido a que permite la identificación de fallos en la implementación, en el funcionamiento, en el rendimiento y permite asegurar la calidad del código final.

### 10.5.1. Plan de pruebas

Al tratarse de un software de monitorización que va a depender de un sistema de adquisición de datos externo implementado en el sistema embebido que ha sido diseñado e implementado en este proyecto, el plan de pruebas sigue un esquema especial puesto las pruebas del software se harán implementando el sistema embebido y descargando su software de control en él.

El plan de pruebas desarrollado consiste en las siguientes fases:

- Pruebas de clases: Tras completar cada clase, se probaban observando que el comportamiento de todos sus método es el adecuado. Se tuvo especial cuida en probar los casos extremos puesto que hay determinadas opciones que necesitan que se cumplan un serie de requisitos para funcionar.

- Pruebas del sistema: Una vez están implementadas todas las clases que componen el sistema, se procede a la prueba de conjunto observando que el funcionamiento del sistema en conjunto es el esperado. El sistema será probado tanto con el sistema embebido funcionando como con un terminal para puerto serie desde el que se simulará el protocolo de comunicaciones.

Las pruebas han sido realizadas en dos momentos específicos durante el desarrollo de este software de control para el sistema embebido.

- Durante la implementación: La razón de probar durante el desarrollo es detectar a tiempo los errores de codificación que puedan surgir sobretodo en la clase de control del puerto serie puesto que es necesario que las comunicaciones no fallen ya que se bloquearía todo el proceso de monitorización.

- Finalizada la fase de implementación: Se observa que el sistema funciona tal y como fue analizado y diseñado en conjunto con el sistema embebido ejecutando su software de control.

### 10.5.2. Diseño de las pruebas

- Durante la implementación: Para probar cada componente de la aplicación se probaron con datos adaptados a los requisitos de funcionamiento. A continuación se comenta las pruebas de cada componente:

  - Puerto serie: Se probó el funcionamiento del puerto serie usando un terminal compatible con comunicaciones serie como puede ser *Putty*. Se probaron los métodos de recepción y envío de información y se simuló el protocolo de comunicación entre la aplicación y el sistema embebido comentado en el capítulo anterior con un conjunto de tramas generadas para tal caso. Las tramas son tanto del tipo estándar como de identificador extendido.
  - Rutina de monitorización: Puesto fue implementada usando hilos, se probó el rendimiento y los tiempos de puesta en marcha y parada de la rutina.
  - Ficheros de log: Se verificó que la información escrita en el fichero se log se correspondía con el formato que se especificó.

- Finalizada la implementación: Fueron probados todos los modos de funcionamiento del sistema. En el modo de monitorización de datos, se simuló su funcionamiento tanto usando una terminal como conectando el sistema embebido y estableciendo comunicación con el.

# Capítulo 11

# Puesta en marcha del sistema de monitorización

En este capítulo se procede a mostrar el aspecto final del sistema de monitorización, el cual incluye tanto el sistema embebido con el software de control como la aplicación de monitorización para ordenadores personales. Se muestra a continuación el sistema monitorizando un vehículo de motor y otra versión con una placa de evaluación simulando una centralita del vehículo y enviando tramas de prueba.

## 11.1. Monitorización de vehículo

Para que el sistema de monitorización implementado en el sistema embebido reciba la señal con el nivel de voltaje adecuado para la FPGA, es necesario un interfaz eléctrico para conectar el monitor-CAN a un bus con otros nodos controladores, como en este caso será el coche.

Es necesario una adaptación de niveles eléctrico y una conversión de la señal diferencial a simple a nivel LVTTL (3.3V). Para ello existen circuitos integrados como el MCP2551 de Microchip. El MCP2551 es un dispositivo que sirve como interfaz entre un controlador CAN y el bus físico. Este chip tiene la capacidad de transmitir y recibir diferencialmente siendo compatible con el estándar iso-11898,puede operar a velocidades de hasta 1Mbps.

En las siguientes figuras se puede observar como queda el sistema montado en un vehículo y capturando tramas con la aplicación de monitorización para ordenadores personales. Además se puede observar el chip encargado de convertir la señal diferencial en una simple y el conector OBD-2 para recibir la señal del bus CAN.

Figura 11.1: Sistema de monitorización instalado en vehículo (foto 1)

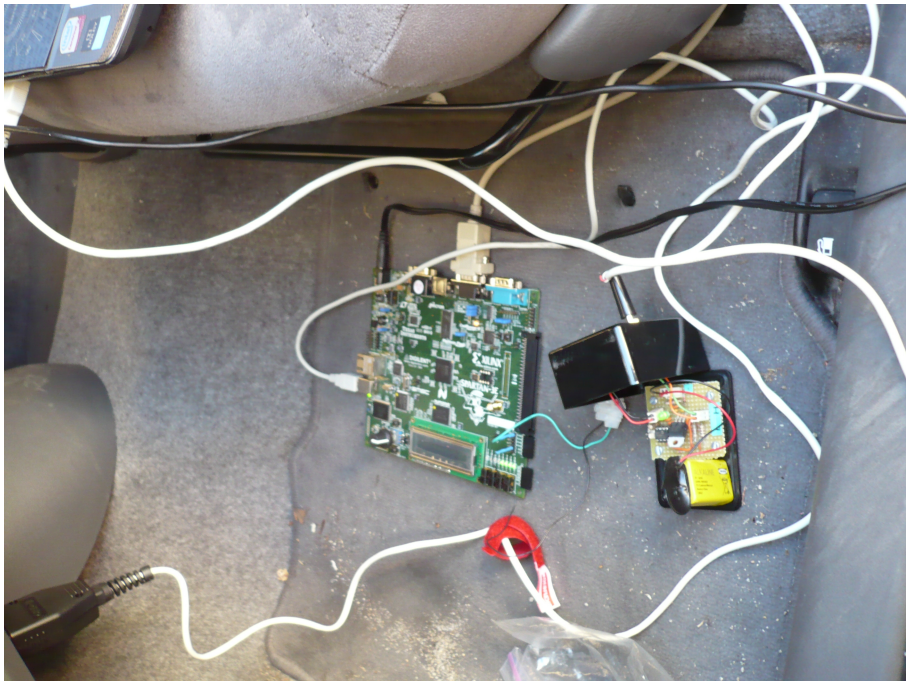

Figura 11.2: Sistema de monitorización instalado en vehículo (foto 2)

## 11.2. Monitorización con la placa de evaluación simulando centralita CAN

El sistema funciona de manera idéntica que en el anterior apartado excepto que esta vez no irá conectado a un coche y que por tanto no será necesario el chip para convertir la señal diferencial en una simple puesto que ambas placas de evaluación son idénticas y por tanto trabajan con los mismos niveles de voltaje (LVTTL).

A continuación se muestra como queda el sistema montado con ambas placas de evaluación y un PC ejecutando la aplicación de monitorización.

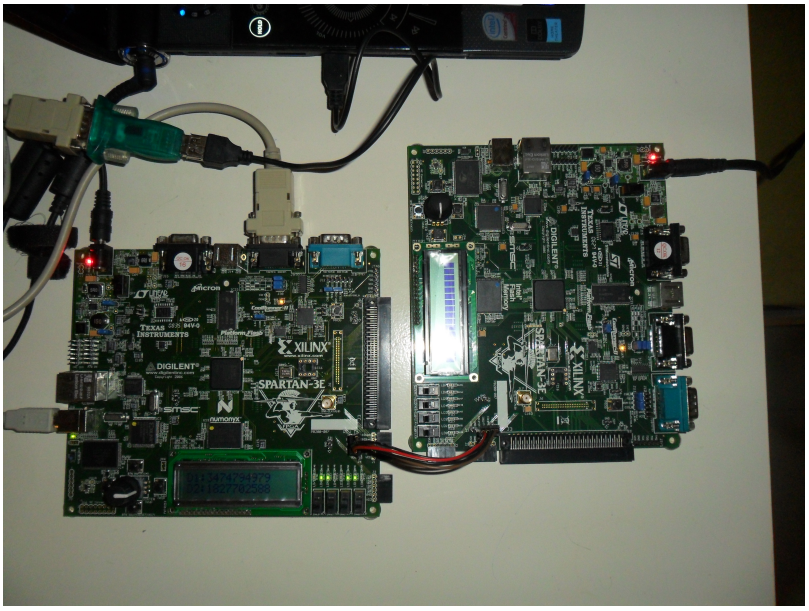


Figura 11.3: Sistema de monitorización con dos placas de evaluación Spartan 3E Starter Kit

Se puede observar que la conexión entre ambas se realiza con dos cables, uno que transmitirá la señal y otro de tierra, desde la placa que simula la centralita a la que tiene implementado el sistema de monitorización. Se observa además el formato de los datos de la trama mostrados por la pantalla LCD.

En la siguiente figura se observa que está en funcionamiento el modo de transmisión de datos a la aplicación de monitorización que está siendo ejecutada en el PC debido a que están parpadeando los ocho leds de la placa de evaluación.

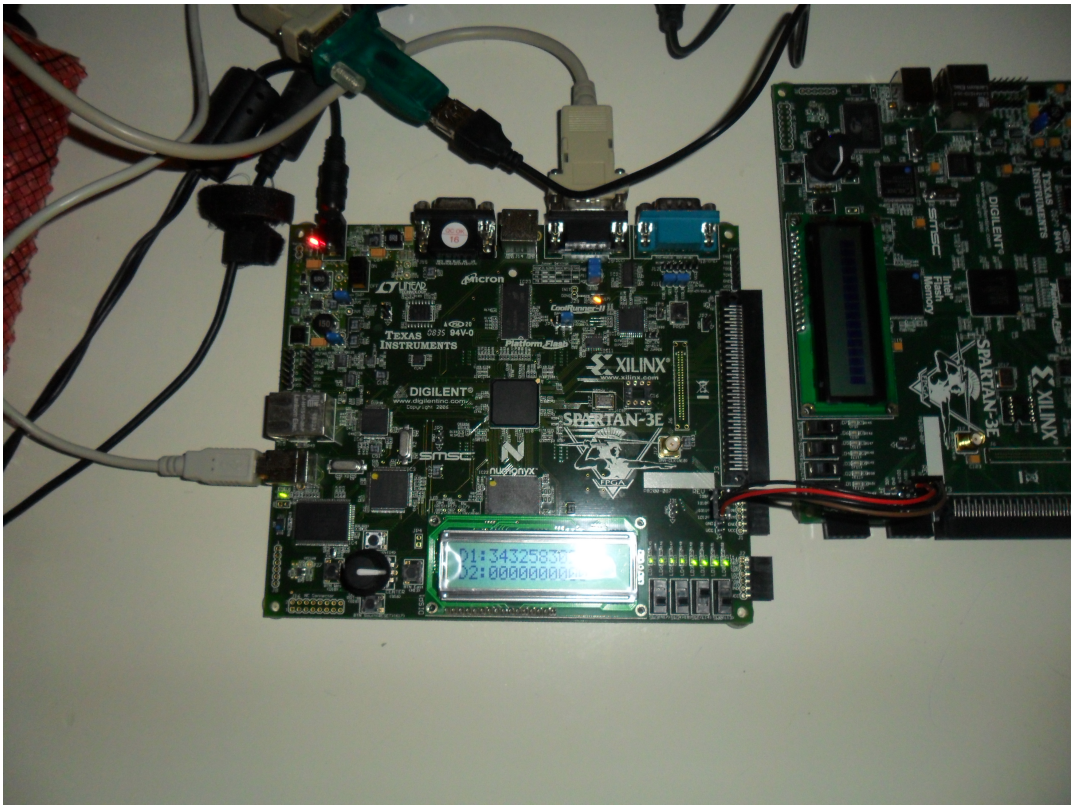


Figura 11.4: Sistema de monitorización con dos placas de evaluación Spartan 3E Starter Kit (Modo transmisión)

A continuación se muestra una captura del programa ejecutándose en un PC en modo monitorización:

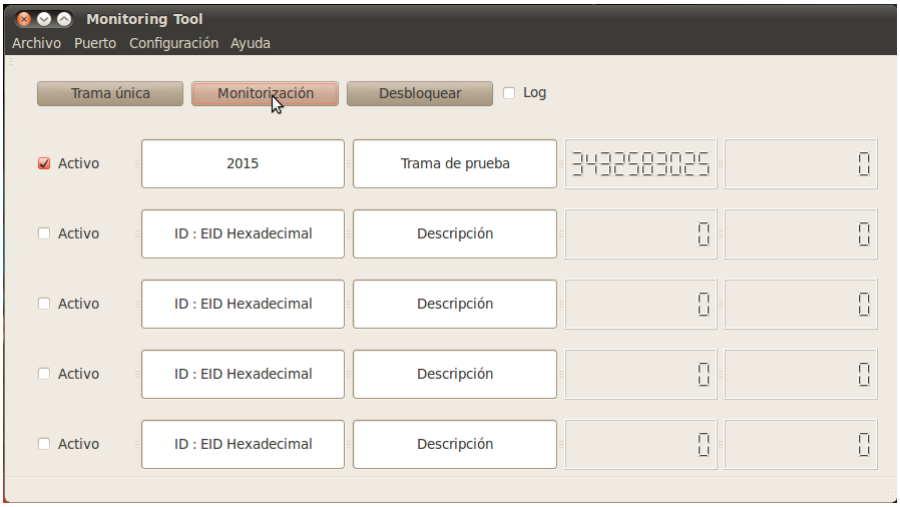


Figura 11.5: Sistema de monitorización con dos placas de evaluación Spartan 3E Starter Kit (Monitoring Tool)

Por último se muestra una parte de un log capturado de un Toyota Prius:

```
288::8::0:270533725:jue jul 14 19:34:00 2011

179::6::0:4554:jue jul 14 19:34:00 2011

35::7::33489407:42:jue jul 14 19:34:00 2011

57::4::1157630861:0:jue jul 14 19:34:00 2011

58::7::0:2412083:jue jul 14 19:34:00 2011

32::3::7:0:jue jul 14 19:34:00 2011

610::4::1223935:0:jue jul 14 19:34:00 2011

610::4::65641:0:jue jul 14 19:34:00 2011

48::8::2214592512:2097372:jue jul 14 19:34:00 2011

59::5::721107:30:jue jul 14 19:34:00 2011

610::4::65641:0:jue jul 14 19:34:00 2011

580::8::268435456:94:jue jul 14 19:34:00 2011

580::8::268435456:94:jue jul 14 19:34:00 2011

37::8::593918:2021161131:jue jul 14 19:34:00 2011

840::6::65536:82:jue jul 14 19:34:00 2011

610::4::65641:0:jue jul 14 19:34:00 2011

57::4::1157630861:0:jue jul 14 19:34:00 2011

35::7::33554943:44:jue jul 14 19:34:00 2011

57::4::1157630861:0:jue jul 14 19:34:00 2011

969::8::67051778:2181236613:jue jul 14 19:34:00 2011

32::3::7:0:jue jul 14 19:34:00 2011

58::7::0:2412083:jue jul 14 19:34:00 2011

840::6::65536:82:jue jul 14 19:34:00 2011
```

```
179::6::0:4554:jue jul 14 19:34:00 2011

56::7::3221227520:7:jue jul 14 19:34:00 2011

62::3::1223935:0:jue jul 14 19:34:00 2011

56::7::3221227520:7:jue jul 14 19:34:00 2011
```

# Capítulo 12

# Conclusiones

En este capítulo se expone tanto las valoraciones personales como las técnicas así como las posibles ampliaciones que pueden realizarse sobre este proyecto final de carrera.

## 12.1. Valoración personal

El desarrollo de este trabajo ha supuesto el primer proyecto completo que he realizado a lo largo de la carrera. Debido a la temática tan variada sobre la que trata este proyecto he podido poner en práctica todos los conocimientos que he ido adquiriendo a lo largo de la carrera.
He conseguido terminar el proyecto en el tiempo que me propuse que era de un año desde el comienzo del tercer curso de la carrera. El límite de un año venía de mi intención de acabar la carrera en tres años exactos y poder comenzar el segundo ciclo inmediatamente. Esto ha significado que he tenido que desarrollar y terminar todo este proyecto en paralelo al tercer curso, en el año académico 2010/2011, lo cual es una tarea bastante dura teniendo en cuenta que tenía que sacar adelante 11 asignaturas, un curso de inglés para obtener el nivel B1 y el proyecto.

En mi opinión se trata de un proyecto muy completo y complejo debido a que se tratan gran cantidad de partes de la informática, algunas de ellas poco tratadas durante la carrera, lo que me ha obligado a investigar y aprender por mi cuenta buscando y leyendo libros, datasheets, white papers, etc... sobre el tema y por supuesto en inglés. Los temas tan diversos que he tratado en este proyecto han sido los siguientes:

- Protocolos de comunicaciones.
- Electrónica.
- Desarrollo de hardware mediante descripción VHDL.
- Simulación de diseños electrónicos.
- Desarrollo de sistemas embebidos.
- Programación de sistemas embebidos y microcontroladores.
- Desarrollo de aplicaciones con interfaz gráfica.

- Generación de Logs.

La parte más dura ha sido la del diseño del periférico receptor de tramas basado en el protocolo CAN. Es muy difícil desarrollar desde cero un receptor para un determinado protocolo y para llevarlo a cabo he necesitado varios meses de estudio del protocolo, leyendo los documentos del estándar y de los principales fabricantes de nodos CAN, para entender sus funcionamiento y formato de transmisión de información. Todo esto hay que trasladarlo a un diseño hardware para implementarlo sobre una FPGA.

A lo largo de la carrera solo hay una asignatura que trata sobre FPGA y VHDL que introduce muy bien los conceptos del lenguaje VHDL y el desarrollo de pequeños diseños. Ha servido como base para seguir formándome en este aspecto y ampliarlo para llevar a cabo un diseño mucho más complejo. Estoy satisfecho por haber completado el desarrollo de un componente hardware complejo y que éste funcione correctamente según el estándar.

Con respecto a los sistemas embebidos, ha sido todo un reto implementarlo y programarlo ya que para ello he tenido que aprender a manejar el entorno de desarrollo de Xilinx, el cual no conocía y que es muy utilizado en la industria electrónica. Ver como sistema embebido que has desarrollado e implementado desde cero funciona tal y como estaba planificado y que se integra bien con el periférico receptor que también he desarrollado es muy satisfactorio.

Por último está el tema de los programas con interfaces gráficas. En la carrera no hay ninguna asignatura obligatoria que trate sobre este tema y la única optativa que lo trata no la he cursado. Me hacía bastante ilusión desarrollar el núcleo de la aplicación, el cual tenía que ser capaz de comunicarse con el sistema embebido desarrollado en este proyecto, y luego desarrollar una interfaz que hiciese que su uso fuese muy sencillo por parte del usuario.

En cuanto a los lenguajes de programación, este proyecto me ha ayudado a dominar todavía mas el lenguaje C++ que he ido aprendiendo durante la carrera y que cada vez le saco más provecho.

En cuanto al VHDL puedo decir que he ampliado mucho mis conocimientos y que conozco más en profundidad los entresijos de este lenguaje de descripción de hardware con sus especiales particularidades a la hora de elegir la forma de codificación que haga más óptimo el diseño final.

Desde el principio tenía la idea de hacer que todo lo desarrollado fuera software libre. Este objetivo se ha cumplido en el diseño del sistema embebido ya que aunque hay alternativas libres, aumentaban excesivamente el tiempo de desarrollo. La idea de implementar el sistema embebido usando el entorno de desarrollo es que todo esto sirva como referencia para la futura asignatura sobre sistemas embebidos que será impartida en tercero del nuevo grado en informática.

Por último me gustaría añadir que no quiero imaginar lo difícil que me hubiera sido escribir este documento si no hubiera contado con la ayuda del lenguaje LaTeX, el cual aprendí gracias a un taller que fue ofertado durante la Semana de la Ciencia del año pasado y que automatiza la generación del documento, controla el formato, etc...

## 12.2. Valoración técnica

Después de dar mi punto de vista personal, a continuación voy a detallar una serie de conclusiones objetivas sobre este proyecto:

- El documento de especificación de requisitos que se elaboró al comienzo de este proyecto se ha cumplido estrictamente en cada una de las cinco fases de las que se ha compuesto el proyecto.
- El periférico receptor de tramas basado en el protocolo CAN es compatible con todo tipo de sistemas y puede ser usado por cualquier desarrollador de hardware debido a que además de tener licencia de software libre, está alojado en una forja de software pública.
- El sistema embebido desarrollado ha cumplido los requisitos de rendimiento y recursos ocupados de la FPGA.
- La aplicación de monitorización junto con el sistema embebido alcanzan un rendimiento más que aceptable en cantidad de tramas recibidas teniendo en cuenta que se pierde una cierta cantidad de tiempo en establecer la comunicación entre el sistema embebido y la aplicación de monitorización. La comunicación es muy robusta siendo prácticamente imposible que se produzca un fallo de comunicación.

## 12.3. Ampliaciones futuras

Este proyecto tiene una gran cantidad de lineas de ampliación. Paso a enumerarlas:

- El sistema de monitorización sólo implementa un periférico receptor de tramas. Puesto que en la FPGA sobran recursos, se puede aumentar el número de receptores de tramas para monitorizar más de un bus. Para ello tal vez sería necesario implementar un sistema embebido con más de un procesador y que estos a su vez lanzasen hilos de ejecución en paralelo.
- El procesador Microblaze es capaz de ejecutar un núcleo de GNU/Linux denominado $\mu$Linux. Como sistema operativo que es, ampliaría enormemente las capacidades del sistema embebido. El núcleo de Linux podría mostrar la información que es mostrada por el LCD de la placa de evaluación por el puerto VGA. Con esto, se podría mostrar la información añadiendo gráficos e información más visual que los propios números solos.
- El creciente aumento en el rendimiento de los dispositivos de móviles haría posible la comunicación del sistema embebido con un dispositivo móvil usando Bluetooth y mostrar la información de monitorización en una aplicación para el propio dispositivo móvil.

# Apéndice A

# Software utilizado

## Doxygen

Esta herramienta realiza una documentación automática de código fuente. Se utiliza para generar la documentación del código fuente que compone el proyecto final de carrera. Puede generar esta documentación en varios formatos, y entre ellos, LaTeX, de forma que podemos utilizar ese código generado en nuestra memoria de forma automática. Otro de los formatos interesantes de HTML puesto que puede ser subido a una web o una forja de software libre.

## Dia

*Dia* es un editor de gráficos vectoriales el cual incluye distintas plantillas para distintos tipos de gráficos, como pueden ser UML, ERe, diagramas de flujo, esquemas Cisco de red y un larguísimo etcétera.

Estos diagramas podemos exportarlos a diversos formatos de imagen (`.png`, `.eps`, ...) o a formato `.tex`.

## Xilinx Project Navigator

*Xilinx Project Navigator* es una herramienta desarrollada por Xilinx para la síntesis y análisis de diseños HDL, la cual permite al desarrollador sintetizar ("compilar") sus diseños, realizar análisis de tiempo, examinar diagramas RTL, simular un diseño y configurar el dispositivo de destino en el que se implementará el diseño.

Es un programa de código cerrado y sólo compatible con FPGAs de Xilinx. Dispone de varios tipos de licencias entre las que encontramos la "Web pack" que permite usar el entorno gratis. La limitación de esta licencia es que sólo permite sintetizar e implementar diseños que tengan como destino una FPGA de la familia Spartan.

## Isim

*Isim* es el simulador integrado en el entorno *Project Navigator*. Permite hacer simulaciones de los diseños en cuatro fases distintas del proceso de implementación: Behavioral, Post-Translate, Post-Map, Post-Route.

Al igual que el *Xilinx Project Navigator* dispone de licencia gratuita y de pago diferenciándose en la precisión con la que miden el tiempo.

## EDK

*Embedded Development Kit* es el entorno de desarrollo integrado para sistemas embebidos desarrollado por Xilinx. Permite diseñar e implementar sistemas embebidos para FPGAs de Xilinx basados en los soft-cores Microblaze y Power PC.

Gracias a su interfaz gráfica es posible diseñar un sistema embebido completo en un tiempo muy reducido, añadir periféricos tanto propietarios de Xilinx como periféricos diseñados por el usuario y escritos en VHDL, diseñar el mapa de memoria del sistema y configurar el procesador del sistema en función de las necesidades de rendimiento/espacio del sistema.

Es necesaria una licencia de pago para poder usar este entorno de desarrollo.

## SDK

*Software Development Kit* es el entorno de desarrollo de software para sistemas embebidos desarrollado por Xilinx. Permite programar los sistemas embebidos para FPGAs de Xilinx basados en los soft-cores Microblaze y Power PC que han sido diseñados y exportados desde el *EDK*.

Entre sus funcionalidades más destacadas está la compilar y generar el script del linker para el software que se desea ejecutar en el procesador del sistema embebido. Permite además descargar el diseño del sistema embebido junto con el software en la FPGA. Está basado en el IDE *Eclipse* y usa el compilador gcc.

Es necesaria una licencia de pago para poder usar este entorno de desarrollo.

## CrossWorks for ARM

Entorno de desarrollo de Rowley Associates para codificar y depurar programas para microcontroladores con núcleo ARM. Permite descargar y depurar el código en el microcontrolador mediante JTAG. Soporta los lenguajes de programación C y C++. Incluye una biblioteca de rutinas para software de control de microcontroladores.

# Qt 4 + Qt Creator

*Qt 4* es una biblioteca para interfaces gráficas con licencia LGPL desarrollada por la división de software Qt de Nokia.

Es compatible con todas las principales plataformas, y tiene un amplio apoyo. El API de la biblioteca cuenta con métodos para bases de datos SQL, XML, hilos, red, manipulación de ficheros y estructuras de datos. Como lenguaje de programación nativo soporta C++ aunque a través de bidings puede ser usada con otros lenguajes de programación

Distribuida bajo los términos de GNU Lesser General Public License (y otras), Qt es software libre y de código abierto.

*Qt Creator* es el entorno de desarrollo para la biblioteca Qt. Incluye un creador de interfaces gráficas llamado *Qt Designer* que permite diseñarlas de forma visual y exportarlas a formato XML que posteriormente usará *Qt Creator* para generar el código C++ correspondiente a la interfaz. Permite la creación de aplicaciones basadas en *Qt 4* de manera sencilla y rápida.

# Apéndice B

# Manual de usuario e instalación

## B.1. Compilación de la biblioteca Qextserialport

Para poder compilar la aplicación de monitorización debemos tener instalada la biblioteca *Qt* y la IDE *Qt Creator*. En este manual se supondrá que ya se encuentran instaladas en el ordenador del usuario.

Lo primero que debemos hacer es obtener la biblioteca Qextserialport. Para ello debemos instalar una herramienta de control de versiones llamada *Mercurial* que podemos encontrar en su sitio web[1]

Una vez instalado *Mercurial*, abrimos un terminal y nos situamos en el directorio donde queremos descargar el código fuente de la biblioteca Qextserialport. Ejecutamos el siguiente comando para descargar la biblioteca.

```
hg clone https://qextserialport.googlecode.com/hg/ qextserialport
```

Ahora desde el directorio de la biblioteca podemos compilar de dos formas distintas:

### B.1.1. Desde Qt Creator

Simplemente abrimos el fichero qextserialport.pro, que encontramos en el directorio donde descargamos la biblioteca, desde el IDE *Qt Creator* y pulsamos en la flecha verde o en Construir ->Reconstruir todo.

[1] http://mercurial.selenic.com/

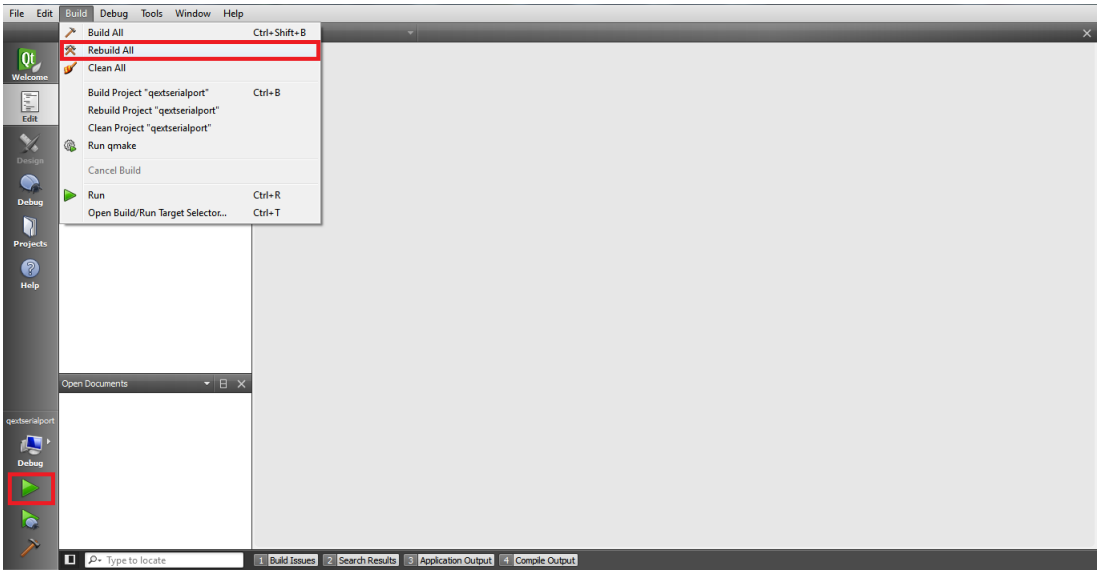


Figura B.1: Compilación de la biblioteca

### B.1.2. Desde el terminal

Abrimos un terminal, nos situamos en el directorio donde esté la biblioteca y ejecutamos los siguientes comandos:

`qmake qextserialport.pro`
Con este comando se procesa el fichero del proyecto generando el conjunto de makefiles y directorios para poder compilar el proyecto con el compilador de C++.

Si estamos en GNU/Linux, compilamos como sigue:
`make`

Si estamos en Windows, compilamos con el siguiente comando:
`mingw32-make`

## B.2. Instalación de la biblioteca

### B.2.1. En GNU/Linux

En GNU/Linux se generan los siguientes cuatro ficheros ficheros:

- libqextserialportd.so
- libqextserialportd.so.1
- libqextserialportd.so.1.2
- libqextserialportd.so 1.2.0

Estos archivos deben ser copiados en el directorio /usr/lib. Una vez copiados la biblioteca queda instalada.

### B.2.2. En Windows

En Windows se generan un único fichero:

- libqextserialportd1.dll

Este fichero deben ser copiados en el directorio Windows o en el del propio programa. Una vez copiado la biblioteca queda instalada.

## B.3. Compilación de la aplicación de monitorización

Para compilar la aplicación basta con seguir los mismos pasos que al compilar la biblioteca Qextserialport. Lo podemos consultar en el apartado B.1. La biblioteca Qextserialport debe estar instalada para compilar la aplicación.

## B.4. Uso de Monitoring Tool

El uso de *Monitoring Tool* es muy sencillo debido a la amigable interfaz de usuario de la que dispone.

En la interfaz de usuario podemos encontrar los siguientes elementos/opciones:

1. Menús desde los que podemos configurar diversas opciones del puerto de comunicaciones y seleccionar el fichero de log.
2. Modo trama única: Pide una trama al sistema embebido que está ejecutando el software de monitorización.
3. Modo monitorización: Ejecuta una monitorización continua pidiendo tramas al sistema embebido que está ejecutando el software de monitorización.
4. Desbloquea el sistema embebido en caso de fallo en las comunicaciones.
5. Activa el modo log.
6. Activa el visualizador.
7. Diálogo para introducir los identificadores de las trama que queremos visualizar en ese visualizador.
8. Diálogo para introducir la descripción de la trama de ese visualizador.
9. Dos displays LCD para visualizar los 8 bytes de datos de las tramas.

Figura B.2: Interfaz de Monitoring Tool


Figura B.3: Diálogo de configuración del puerto serie

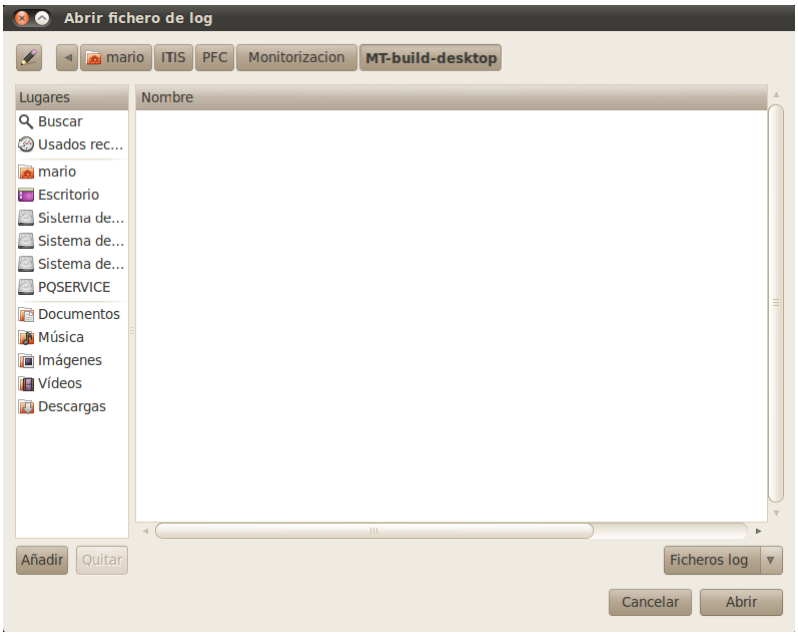


Figura B.4: Diálogo para introducir el fichero de log

# Bibliografía


[1] Pérez, S.A. , Soto, E. , Fernández, S. Diseño de Sistemas Digitales con VHDL. Thomson 2002

[2] Ron Sass, Andrew G. Schmidt. Embedded systems design with platform FPGAs. Morgan Kaufmann Publishers. 2010

[3] Videotutorial EDK http://www.youtube.com/user/SILICAMarcom

[4] Standard protocolo CAN http://www.semiconductors.bosch.de/media/pdf/canliteratur/can2spec.pdf

[5] Información protocolo CAN http://www.softing.com/home/en/industrial-automation/products/can-bus/

[6] Guía de referencia Microblaze http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/mb_ref_guide.pdf

[7] Bibliotecas y Sistemas Operativos para sistemas embebidos de Xilinx http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/oslib_rm.pdf

[8] Manual de la placa de evaluación Spartan 3E Starter Kit http://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf

# GNU Free Documentation License

Version 1.3, 3 November 2008
Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

<http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

## Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.
This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.
We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

## 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.
A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.
A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly

within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

## 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.
You may also lend copies, under the same conditions stated above, and you may publicly display copies.

## 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.
If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.
If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.
It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

## 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.
You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.
You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.
The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

# 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.
The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.
In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

# 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.
You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

# 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.
If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

# 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.
If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

# 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.
However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.
Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.
Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

## 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/. Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

## 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.
"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.
"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.
An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.
The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

## ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.
If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.